Analysis of Low-Level Implementations of Cryptographic Protocols

Andriana Evgenia Gkaniatsou

Doctor of Philosophy
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh
2018

Abstract

This thesis examines the vulnerabilities due to low-level implementation deficiencies of otherwise secure communication protocols in smart-cards. Smart-cards are considered to be one of the most secure, tamper-resistant, and trusted devices for implementing confidential operations, such as authentication, key management, encryption and decryption for financial, communication, security and data management purposes. The self-containment of smart-cards makes them resistant to attacks as they do not depend on potentially vulnerable external resources. As such, smart-cards are often incorporated in formally-verified protocols that require strong security of the cryptographic computations. Such a setting consists of a smart-card which is responsible for the execution of sensitive operations, and an Application Programming Interface (API) which implements a particular protocol. For the smart-card to execute any kind of operation there exists a confidential low-level communication with the API, responsible for carrying out the protocol specifications and requests. This communication is kept secret on purpose by some vendors, under the assumption that hiding implementation details enhances the system's security. The work presented in this thesis analyses such low-level protocol implementations in smart-cards, especially those whose implementation details are deliberately kept secret. In particular, the thesis consists of a thorough analysis of the implementation of PKCS#11 and Bitcoin smart-cards with respect to the low-level communication layer. Our hypothesis is that by focusing on reverse-engineering the low-level implementation of the communication protocols in a disciplined and generic way, one can discover new vulnerabilities and open new attack vectors that are not possible when looking at the highest levels of implementation, thereby compromising the security guarantees of the smart-cards.

We present REPROVE, a system that automatically reverse-engineers the low-level communication of PKCS#11 smart-cards, deduces the card's functionalities and translates PKCS#11 cryptographic functions into communication steps. REPROVE deals with both standard-conforming and proprietary implementations, and does not require access to the card. We use REPROVE to reverse-engineer seven commercially available smart-cards. Moreover, we conduct a security analysis of the obtained models and expose a set of vulnerabilities which would have otherwise been unknown.

To the best of our knowledge, REPROVE is the first system to address proprietary implementations and the only system that maps cryptographic functions to communication steps and on-card operations. To that end, we showcase REPROVE's usefulness to a security ecosystem by integrating it with an existing tool to extract meaningful state-machines of the card's implementations. To conduct a security analysis of the results we obtained, we define a threat model that addresses low-level PKCS#11 implementations. Our analysis indicates a series of implementation errors that leave the cards vulnerable to attacks. To that end, we showcase how the discovered vulnerabilities can be exploited by presenting practical attacks.

The results we obtained from the PKCS#11 smart-card analysis showed that proprietary implementations commonly hide erroneous behaviours. To test the assumption that the same practice is also adopted by other protocols, we further examine the low-level implementation of the only available smart-card based Bitcoin wallets, LEDGER. We extract the different protocols that the LEDGER wallets implement and conduct a through analysis. Our results indicate a set of vulnerabilities that expose the wallets as well as the processed transactions to multiple threats. To that end, we present how we successfully mounted attacks on the LEDGER wallets that lead to the loss of the wallet's ownership and consequently loss of the funds. We address the lack of well-defined security properties that Bitcoin wallets should conform to by introducing a general threat model. We further use that threat model to propose a lightweight fix that can be adopted by other, not necessarily smart-card-based, wallets.

Acknowledgements

I was was never good at putting my thoughts on paper but I will try to do it now, as there is a set of people that I should thank for supporting me during my PhD life. Of course the list is not exhaustive and I should mention that those that I missed are not forgotten.

I would like to thank my advisors Prof. Alan Bundy and Dr Fiona McNeill for trying their best to understand my work although they come from a completely different background, their support and help throughout these years. I would also like to thank Dr. Graham Steel for his helpful insights on my work. Besides my advisors, I am very grateful to Dr Myrto Arapinis who has helped me, provided fruitful advice, supported and encouraged me both in my work but also as a friend. Myrto you have been wonderful, thank you for everything. I would also like to thank Prof. Aggelos Kiayias, the Dream group and the security group for their useful feedback.

Of course I owe a huge thank you to my parents and especially to my mother. Despite all the difficulties, she has been giving me her unconditional love and support all these years. She has always stood by my side and encouraged me. She visited me in Edinburgh during stressed periods just to keep me sane, despite many obstacles. I cannot express how much I owe to her. Thank you for everything mum, I could not have done it without your support.

I would like to express my deepest gratitude to my dearest friends, both in Greece and in Edinburgh, for keeping me sane all these years, for their love, for their support, for the great laughs and for all these wonderful moments that made these years unforgettable. Eirini P., Vasiliki K., Konstantina D., Zisis P., Evangelia P., Mary I., Elena T., Gelly A., and Eleni K., thank you for everything. A special thanks to Eleni Papathomaidou for being my inspiration and making me love Computer Science.

Lastly but most importantly, I owe my deepest gratitude to the person who supported me the most, my husband Prof. Stratis Viglas. He has been by my side since the first moment of this journey, advised me, encouraged me and made me see the bright side of things. He has spent numerous hours listening to my work problems trying to understand my work and provide fruitful feedback, and even more hours listening to me complaining. He has been my mentor, my advisor, my friend, my family. Stratis from the bottom of my heart, thank you for everything. I could not have done this without you.

Finally, I would like to to thank EPSRC for funding my work.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the chapters that appear in this thesis have been published in the following papers:

- Gkaniatsou, A., Arapinis, M., and Kiayias, A. (2017) Low-Level Attacks in Bitcoin Wallets. In *International Security Conference*, pages: 233-253.
- Gkaniatsou, A., McNeill, F., Bundy, A., Steel, G., Focardi, R., and Bozzato. C. (2015) Getting to know your Card: Reverse-Engineering the Smart-Card Application Protocol Data Unit. In *Annual Computer Security Applications Conference*, pages: 441-450.

(*Andriana Evgenia Gkaniatsou*)

Table of Contents

Chapter 1

Motivation

Smart-cards, also called integrated circuit cards (ICC), are cryptographic hardware with an embedded circuit that offer an hermetic and isolated environment for data storage and management. Due to their physical characteristics they are often considered to be tamper-resistant: the intended functionality of the hardware and the data that it manages is protected from malicious use and cannot be tampered with. Smart-cards enable the generation, storage, transformation, cryptographic manipulation, and protection of sensitive data. They are therefore primarily used as a proven mechanism to offer security services by a vast array of applications.

The first integrated circuit cards were patented back in 1968 by Jurgen Dethloff and Helmut Grotrupp, but it was not until 1974 that the first smart-card was developed by the French inventor Roland Moreno, also known as 'the Father of the Microchip'. Since their early development years, smart-cards have been widely adopted and have been used in many diverse sectors. For instance, in 1987 Turkey launched smart-card based driver licences; the first electronic prepaid card was introduced in Denmark in 1992; while 1994 marked the first official specification of smart-cards for bank usage by the consortium of Europay, MasterCard, and Visa, also known as EMV.

Ever since, the design and capabilities of smart-cards have branched and evolved to offer low-cost solutions for complex scenarios encompassing single- and multi-application smart-cards. The applications of smart-cards are very diverse and spread from simple user authentication, to data access protection, to financial processing. They are increasingly adopted for many commercial applications, such us identification cards, telecommunication, access control systems, banking, to name but a few examples. Indeed, they have become such a tightly integrated part of our everyday lives as to be considered commodity. All the example applications we listed incorporate smart-cards to generate, store, process, and/or perform operations over data of a sensitive nature.

Nowadays, smart-cards are becoming the de-facto standard of assured security and their uses range from the highly commercial to the inherently private and user-centric. For example, CERN, the European Organisation for Nuclear Research, incorporates smart-cards for securing access to their mission-critical systems; while Facebook users may employ smart-cards to authenticate their access to the social networking service. But the use of smart-cards has far exceeded data access and user authentication. The current emerging trend is to incorporate smart cards into data processing by means of enabling them to execute sensitive operations. Protocols that offer increased security coverage are being implemented solely on smart-cards that not only store and provide access to static data, but also manipulate data in a tamper-proof way.

The increased adoption and adaptability of smart-cards creates the need for specifying universal mechanisms and security requirements that all smart-card should adhere to. Doing so creates a base-line that all developers can safely assume the existence of, and thus build their applications without needing to design custom smart-cards. That way, improvements in smart-card technology can benefit all applications as they all have a common interface. Creating that base-line, however, is far from a trivial exercise as smart-cards themselves have unique characteristics, the most important of which are their limited memory and computing power as well as their reliance on an external device, the smart-card reader, to ensure their operation. Any application incorporating smart-cards has therefore two tiers (the smart-card and its reader) and requires some careful thinking in order to be efficiently specified and correctly used.

Securing a system from potential threats has long been a central aspect in dealing with sensitive data. Computational security has emerged as a disciplined way to enable many mechanisms that can be proven beneficial when designing and implementing a system managing sensitive data. Work on computational security is very diverse and includes the development of threat modelling techniques to capture attacks to data; to reasoners where the particular aspects of a system are abstracted and studied at a higher level to prove its correctness or identify its vulnerabilities; to standardisations that define the criteria under which particular protocols are considered secure.

Although such general mechanisms have become common practice and are easily accessible to the interested parties, it is common for commercial vendors to follow a "security through obscurity approach": they provide security by eschewing standard conformance and hiding the underlying implementation details. Such practice provides some levels of security as it requires from the adversary to spend a substantial amount of time to reverse-engineer and analyse the implementation in order to attack it. However, such practice has been repeatedly criticised by security experts and researchers as it cannot not guarantee a secure setting.

A common perception is that by proving the correctness and the security of the high-level protocol that a smart-card implements is enough of a proof that the same properties transcend to the application and use-case of the smart-card and indeed the entire system that makes use of the protocol. In other words, proving that the high-level code that implements a data exchange protocol between a smart-card and its reader satisfies certain security constraints, has been traditionally considered equivalent to proving that all layers below that implementation of the protocol are also secure. The reasoning is that all lower levels are abstracted by the implementation and are thus not exposed. This thesis challenges that assumption and shows that by studying the potential low-level implementations of security protocols in a systematic way by means of reverse-engineering we can identify new vulnerabilities and open new attack vectors that cannot be considered by verifying the high-level protocols alone.

Assessing the security of smart-cards in practice. The way smart-cards are designed and intended to be used implies a dependency on the card communicating with a host device, commonly referred to as a reader. That communication is responsible for initialising any kind of operation and data transfer to/from the card. Any operations on the card dictate the communication between card and reader be deliberately confidential. The central notion of work on smart-card security is ensuring that this communication remains secure under potential threats. If for any reason this communication is vulnerable and can be compromised the whole purpose of smart-cards being tamper-resistant environment is defeated. There is a gap in literature when it comes to having well defined security requirements with regard to the communication layer. This shortcoming, which results in the inability for one to reason about the security properties of the smart-card at a higher-level coupled with the common practice of proprietary implementations often results in flaws that render the implementation itself and the card by association prone to attacks. Verifying complex implementations, especially when these are treated as a black box, is a particularly difficult task as it requires increasing the level of abstraction in order to be able to reason about the implementation; the development of well-defined models that map to the higher level of abstraction; a deep understanding of the potential risks involved; and standardised ways of translating both (a) instances of different implementations to the higher level of abstraction and (b) security risks to verifiable scenarios over the abstraction.

Though smart-cards aim to increase the security of a system by providing an isolated and protected environment, they are also easily recognisable and prominent system components, which makes them ideal targets to malicious users. Low-level smart-card communication was traditionally somewhat overlooked by vendors, which lead to flaws and vulnerabilities that were soon discovered by security researchers. Currently, the literature contains a multitude of attacks and vulnerabilities: from tools that allow 'sniffing' the communication (*i.e.*, actively observing and intercepting the communication trace) and on-the-fly manipulating the content of the communication (*e.g.*, [Choudary, 2010, De Koning Gans and de Ruiter, 2012])); to various low-level attacks that depend on the kind of smart-card and application domain being targeted (*e.g.*, [Murdoch et al., 2010b, Barbu et al., 2012]). The way an attack is formulated is heavily customised to the use-case, *e.g.*, Personal Identification Number (PIN) or authentication data sniffing, requires a different angle of attack than accessing sensitive keys, or executing unauthorized operations, or cloning the card. At any rate, identifying and materialising potential threats requires a good understanding of the underlying implementation itself in addition to the high-level protocol being implemented.

Use of Security Application Programming Interfaces. Security Application Programming Interfaces (APIs) have the objective of securing access to sensitive resources. The design of such APIs is critical, as they must ensure the secure creation, deletion, import and export of a key from the device. Also, they are responsible for permitting the use of these keys for encryption, decryption, digital signing, and authentication so that if a device operates in a hostile environment and is exposed to malicious software the card's functionality remains secure.

Whenever an API call is made, there is a deliberate communication between the API and the connected smart-card defined by the Application Data Protocol Unit (APDU). The API call is translated into smart-card commands which are sent via the low-level communication layer. Currently, there does not exist a standardised way for implementing that translation. The design and implementation of that translation depends entirely on the vendors, along with the semantics of the communication.

In this thesis we tackle the problems that arise when the overlooked low-level API implementations do not adhere to the security requirements placed by the higher-level protocol. In particular, we study how security APIs translate high-level protocol specifications into low-level smart-card communication and the vulnerabilities that such a translation exposes. As we will show, it is more often than not that this communication opens new attack vectors. To make our work concrete we extract and analyse the underlying implementations of two currently dominant protocols: the RSA PKCS#11 standard and the Bitcoin protocol.

Targeting real-world applications through the low-level communication. RSA PKCS#11, defines the API for a wide variety of cryptographic devices such as smart-cards, is the most commonly used cryptographic standard. It specifies an ANSI C API that defines mechanisms for accessing sensitive data and implementing cryptographic functions. It aims to isolate an application from the details of the cryptographic device and provide interoperability between devices from different manufacturers.

RSA PKCS#11 API-related attacks were first introduced in [D. Longley, 1992] in the early 1980's, followed by the exposure of the first RSA PKCS#11 vulnerabilities in [Clulow, 2003]. Ever since, there has been considerable work on developing formal techniques for analysing and verifying the correctness of the implementation of RSA PKCS#11. Although the idea of mounting attacks on the security API by exploiting weaknesses on the hardware interface has been introduced since the early 2000's [Bond and Anderson, 2001], little attention has been paid to the low-level implementation of RSA PKCS#11.

Smart-card implementations from different vendors are substantially different and it is common practice to even define proprietary semantics. To conduct a security analysis on that layer, however, it is necessary to understand the semantics of the exchanged messages. To that end, we present a fully automated solution for reverse-engineering the low-level implementation and, by doing so, gaining a full insight into it. We have implemented REPROVE, an automated tool that is implementation-independent and does not require access to the card nor to the API. Given a communication trace, REPROVE infers the semantics of the communication, deduces the card's functionalities, and provides a mapping between the communication layer and PKCS#11 functions. Such tool can be used both by the developers of smart-cards to reason about their implementation, and also by the clients who wish to discover whether their card is secure. We used REPROVE to reverse-engineer seven commercially available PKCS#11 smart-cards and obtain their abstract models. We then leveraged these models to conduct a thorough analysis and identify hidden security flaws that otherwise could not be detected. Thereby, we showcase how the low-level communication layer can be exploited to mount PKCS#11 attacks.

Another application of smart-cards that we have looked at is Bitcoin, the most widely adopted digital currency as of June 2017. Bitcoin is a cryptocurrency and a payment system based on public-key cryptography. The main principles of Bitcoin are: 1. an account is identified by a public key, and 2. to prove ownership of funds, a user must digitally sign a payment with the corresponding private key. Therefore, a salient aspect of Bitcoin is key management: loss of the private keys effectively means loss of funds; and exposure of the public keys conveys privacy loss.

Even though Bitcoin is a digital currency, it is still a financial protocol with real-world implications. As such, and similarly to every financial protocol, there has been a great interest into studying, verifying, attacking, and subsequently proposing solutions for Bitcoin's underlying protocol. To that end, it is highly recommended that the user's keys are stored offline. Currently an emerging trend is incorporating cryptographic hardware due to it's tamper-resistant nature, for managing the account's keys and digitally signing the payments.

While using expensive hardware for the purpose of building a digital wallet is considered to provide security guarantees, often the low-level implementation introduces new attack vectors. In this work we bring into attention how the overlooked low-level protocols that are used to implement supposedly safe hardware wallets can be heavily exploited to mount attacks on the Bitcoin protocol. Our work addresses the implementation of the Ledger Wallets, the only wallets that incorporate smart-cards. We conduct a thorough analysis of the Ledger Wallet communication protocol and show how to successfully attack it in practice. We address the lack of well-defined security properties that Bitcoin wallets should conform to by introducing a general threat model. We further use that threat model to propose a lightweight fix that can be adopted by different technologies.

Contributions and roadmap. In this thesis we address the security properties of the low-level implementations of smart-cards. Our work focuses on two scenarios: PKCS#11 and Bitcoin smart-cards. Apart from the similar nature (*i.e.*, key management and execution of sensitive operations), PKCS#11 are quite diverse in comparison to Bitcoin smart-cards. PKCS#11 smart-cards incorporate all major characteristics defined by the ISO/IEC 7816 standard. For example, apart from executing cryptographic functions, they also perform storage-related operations *e.g.*, read, write, update *etc.*. In contrast, the operations that Bitcoin smart-cards can perform are bound to their application and are only the following two: generation of keys and digital signatures using these keys. Such cards are not designed to manage data objects nor to implement any other operation. The differences between the two categories are substantial, making it infeasible to apply the same methods. As such, our work and consequently this thesis is divided accordingly into parts.

In Chapter 2 we present the necessary general and common preliminaries across the two strands of our work. We introduce the cryptography terminology that we will use throughout the thesis. We then provide the necessary details on ISO/IEC 7816 and show the discrepancies between the inter-industry and proprietary definitions of the commands covered by the standard, and how these discrepancies aggravate the problem of reverse-engineering communication traces. We then further delve into each specific application scenario.

Part One: PKCS#11 smart-cards. The first part of the thesis presents our work on the security analysis of PKCS#11 smart-cards. We discuss the implementation of REPROVE, an automated tool for reverse-engineering smart-cards. Subsequently, we define a threat model for the APDU layer and use that threat model to guide the security analysis of the reverse-engineered smart-cards. Finally we show how REPROVE can be used in concert with other tools that extract state-machines of card implementations to form a larger ecosystem for detailed and diverse security analysis. In more detail, the content of each chapter is as follows.

- Chapter 4 provides the necessary context for our work.

- Chapter 5 provides the details of REPROVE's implementation details and the chosen modelling techniques. A major challenge of reverse-engineering proprietary implementations is tackling and navigating a potentially huge search space. We show that REPROVE manages to narrow-down the search space and extract a meaningful model. We evaluate REPROVE's accuracy on the following seven commercially available smart-cards: Aladdin (now Gemalto) eToken Pro, Athena ASEKey USB, Siemens (now Atos) CardOS V4.3, Atos CardOS V4.4, Atos CardOS V5, RSA SecurID, Safesite TCP ISV1.

- Chapter 6 presents our security analysis of the reverse-engineered smart-cards. To do so, we first introduce a threat model for the APDU layer of PKCS#11 smart-cards. We incorporate the STRIDE [Swiderski and Snyder, 2004] technique to identify potential threat scenarios. We then reduce the attack space by identifying patterns and commonalities between the attacks. The result is a set of the vulnerable data that, if compromised, can lead to a series of mountable attacks. Using these attacks as a guide, we manually analyse the reverse-engineered cards and expose a set of threats. To showcase the importance of obtaining the semantics of the communication and REPROVE's flexibility to co-operate with other systems, we present an ecosystem for extracting meaningful state-machines of the cards' implementations. Finally, and as a case study, we present the feasibility of mounting attacks through the APDU layer by presenting practical attacks on the ATOS Cardos V5 smart-card.

Part Two: Bitcoin Smart-cards. Ledger wallets being the only available smart-card wallets eliminates the need to produce an automated system for reverse-engineering their implementations: we can afford to come up with a focused study. Our work targets: the Nano wallet, which is a USB smart-card, and the Nano S wallet, which is a USB Human Interface Device (HID) smart-card with a screen. We extract and analyse the Ledger wallets, with respect to their APDU implementations. We define a threat model for Bitcoin hardware wallets and show how the Ledger wallets are prone to a series of attacks. We then mount these attacks and prove their feasibility. To rectify the shortcomings of Ledger wallets we propose a lightweight and easily adoptable fix to preserve the privacy and ensure the integrity of the transactions. In more detail:

- Chapter 8 provides the necessary background and context of our work. We give an overview on Bitcoin and its specifications, and present the current state-of-the-art on Bitcoin hardware wallets and related work. We show how the underlying protocol implementation, through hardware-independent in theory, it is is very similar for most wallets in practice.

- Chapter 9 contains our analysis of the extracted Ledger wallet protocols. The Ledger APDU implementation suggest a series of vulnerabilities that allow attacks both on the wallet itself, as well on the Bitcoin transactions.

- Chapter 10 is our empirical proof that the Ledger wallets fail to be secure. We present a series of practical attacks that we have successfully mounted on the Ledger wallets. Our experiments suggested that the wallets fail to secure wallet content or the exchanged information in the context of a transaction: we have managed to tamper with both at will. Moreover, we show how we can successfully disable all the security mechanisms that the Ledger wallet incorporates, through the APDU layer. In doing so, we discover and present serious implementation flaws that allow malicious access to the account's master key, and encryption keys. Based on the threats we have identified, as well as the attacks we have successfully mounted, we suggest a lightweight fix for securing the wallets.

Finally, Chapter 11 summarises our contributions and provides an outlook on the problems in the smart-card area that our work identifies.

Chapter 2

Preliminaries

In this chapter we provide the necessary preliminaries of this thesis. Section 2.1 describes some basic concepts of cryptography with respect to the terminology used throughout this thesis. Our work investigates contact microprocessor smart-cards, which are defined by the ISO 7816 standard. Section 2.2 provides the necessary background on the smart-card communication according to ISO 7816. Section 2.3 discusses the general principles of security protocols and the known attacks.

2.1 Cryptography

One of the main purposes of smart-cards is the practical use of existing cryptographic procedures. Cryptographic algorithms are categorised depending on the keys they use for encryption and decryption, into *symmetric-key* and *asymmetric-key*.

Symmetric algorithms use the same confidential key, known as the *secret key*, for both encryption and decryption. Symmetric algorithms are commonly used for encrypting large volumes of data, as they are in principle designed to be fast. Asymmetric algorithms, also known as a *public-key*, use two mathematically related but not identical keys; a non-confidential key, also known as the *public key*, and a confidential key, also known as the *private key*. Although each public key is related with the corresponding private key, it is computationally infeasible to calculate the private key from the public one. Asymmetric algorithms are commonly used for encrypting small blocks of data as they are slower and require more complex hardware, compared to symmetric ones. Digital signing is another application of asymmetric algorithms.

The upper bound of security that each algorithm provides depends on the size of the used key, given that the security of a system is broken if the key is compromised, as given an $n - bits$ key, the search space of the possible values of that key is 2^n. When defining the security of an algorithm with respect to the size of the key, the complexity of the algorithm is also taken into consideration. Depending on the algorithm used and its complexity, different sizes of keys are used for the same level of security.

The three basic objectives of cryptography is to provide confidentiality, integrity and authenticity of data through different mechanisms. We shall describe some basic concepts of cryptography based on these four objectives.

2.1.1 Algorithms

Confidentiality. *Confidentiality* refers to the concept of ensuring that only authorised entities have access to particular data. Confidentiality is achieved through encryption: a plaintext message m is encoded under a pseudo-random key sk to produce a ciphertext. The process by which the ciphertext is converted back to plaintext, under the decryption key, is called decryption. This mechanism ensures that only the owner of the decryption key can access m. We denote encryption as $enc(m,sk) \rightarrow \{m\}_{sk}$, where m is the plaintext message, sk is the encryption key and $\{m\}_{sk}$ is the resulting ciphertext, and decryption as $dec(\{m\}_{sk},k) \rightarrow m$ where k is the decryption key (depending on the scheme, sk and k can be equal).

Symmetric Encryption. A plaintext message m is encoded by mangling it with the secret key k, $enc(m,k) \rightarrow \{m\}_k$. Decryption of the ciphertext $\{m\}_k$ requires the knowledge of the secret key, $dec(\{m\}_k,k) \rightarrow m$.

Encryption usually takes a fixed size of input and produces a fixed size of output and is categorised into two schemes, *block ciphers* and *stream ciphers*. Block ciphers encrypt a plaintext in blocks of n-bits and then join the encrypted blocks to make the ciphertext. Stream ciphers encrypt a single bit of the plaintext at a time by producing a pseudo-random sequence of bits (*keystream*) which is then used to encrypt each single digit of the plaintext with the corresponding digit of the keystream.

Some of the most commonly used symmetric algorithms are the Triple Data Encryption Standard (3DES) which uses three individual 56-bit keys, Blowfish which has a variable key length from 32 to 448 bits, Twofish which uses 128-, 192- or 256-bit keys, and Advanced Encryption Standard (AES) which uses 128-, 192- or 256-bit keys.

Asymmetric Encryption. A plaintext message m is encoded by mangling it with the public key pk, $enc(m,pk) \rightarrow \{m\}_{pk}$; the ciphertext $\{m\}_{pk}$ can only be decrypted by the owner of the corresponding private key sk, $dec(\{m\}_{pk},sk) \rightarrow m$.

Some of the most commonly used public-key algorithms are the Rivest-Shamir-Adleman (RSA) algorithm which can be used for both encryption and digital signatures and uses 1024- to 4096-bit keys; the Diffie-Hellman algorithm which is the most widely used algorithm for key exchange (discussed in more detail in the following Sections) and has roughly the same key strength as RSA; the Elliptic Curve Cryptography algorithms which are believed to be secure with keys twice the length of equivalent strength symmetric key algorithms.

Integrity and Authenticity. *Integrity* refers to the concept of ensuring that data is in its original representation *i.e.*, it is accurate and unchanged. Authenticity refers to ensuring that data comes from a genuine source. Different cryptographic algorithms exist that ensure data integrity and authenticity, among them are cryptographic hashes, Message authentication Codes and digital signatures.

Cryptographic hash function. Cryptographic hashing is an one-way function commonly used for integrity checking. Hash functions are not based on cryptographic secrets, thus they eliminate the need of sharing secret keys. A hash function takes as input a string of arbitrary length m and outputs a fixed length string, the *message digest* or *digest*. The digest is usually sent along with m to the recipient, who will recompute the hash of m and compared it to the one he received. The output is deterministic and it is computationally infeasible to invert the function. A change of the input results in a change of the output, thus hashes are commonly used in many applications for integrity checking. Hashes can also be used in conjunction with other similar-purpose mechanisms such as the Message Authentication Code (MAC) and the digital signatures.

Message Authentication Code (MAC). MACing is based on symmetric-key cryptography. The MAC algorithm takes as input a secret key k and a message m of arbitrary length, and outputs a MAC value m', also known as a *tag*. Then, m' is appended to the m to form the final message mm', $m+m' \rightarrow mm'$. To validate the authenticity of m, the receiver of mm' must share the same key sk. The recipient recomputes a MAC based on m and his own key and compares it with the received one, m'.

Digital signatures. The main principle of digital signatures is to prove *ownership* of some data *i.e.*, the data was created by the user that claims to have create it. Digital signatures are based on asymmetric-key cryptography and incorporate keypairs: a private key and the corresponding public key. The signature algorithm, in its simplest form, creates a one way hash of the plaintext message, and using a private key *sk*, encrypts that hash. The ciphertext together with the plaintext message compose the signature. We denote the signature process as $sign(m,sk) \to \sigma_{sk}$, where m is the plaintext message, sk the private key and σ_{sk} the resulting signature. The validation of the signature is done using the corresponding public key: the verification algorithm calculates the hash of the plaintext, decrypts the ciphertext included in the signature using the public key and compares the two hashes.

2.1.2 Keys

Keys may have different duration on their usage: they can be designated for long-term usage *e.g.*, a private key for encryption, or short-term usage where the key is destroyed after a designated period *e.g.*, a private session key. Keys are managed by a dedicated *system* which is responsible for their generation, storage, usage, exchange, deletion and replacement. Such management systems also ensure that keys are used correctly by assigning them types, depending on the function they are designated to. In principle, a key should not be used for a different function other than the one of its type. For example, a key being a private signature key defines that this key should only be used for signatures and for no other operations. This policy protects the security of the keys in case that an operation has been compromised. For example, assume a system that uses the same key for signatures and encryption. If the encryption system is weak and the adversary is able to extract the key, then the signature system is also compromised.

2.2 Smart-Cards

Smart-cards are categorised into memory cards, which simply store data and microprocessor cards, which can perform operations over the data in their memory. They are also categorised depending on the type of their interface, into contact and contactless. Contact smart-cards require physical contact with a reader, while contactless have an embedded antenna which allows communication without physical contact. The work presented in this thesis investigates contact microprocessor cards also known as Integrated Circuits Cards (ICC). ICC have an embedded microcontroller with internal memory or a memory chip alone.

Although smart-cards are mainly intended for performing cryptographic computations, their applications can be quite diverse. Smart-cards are commonly used for data storage purposes *e.g.*, health cards and medical insurance cards, for identification

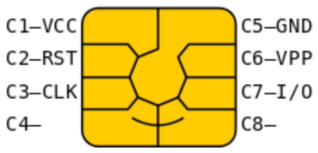


Figure 2.1: The contacts of a smart-card: *VCC* is the power, *RST* is the reset, *CLK* is the clock signal, *GND* is the ground connector, *VPP* is the programming voltage and *I/O* is the input/output connector.

purposes *e.g.*, passports and national identification cards, for communication purposes *e.g.*, GSM and UMTS cards, for electronic payments *e.g.*, credit, debit and Bitcoin cards, and for processing cryptographic computations. Each smart-card has its own operating system, which is responsible for implementing the standard/protocol of the corresponding application. Apart from the cards that are instantiated to a specific application, there are also programmable smart-cards that allow the user to develop and load his own applications on the card. There are various development environments and languages for the programmable smart-cards, with the most commonly used to be the JavaCard, BasicCard and .NET cards. The fundamental characteristics of smart-cards are specified in the ISO 7816 standard.

As defined in ISO 7816-2, smart-cards consist of five contacts from which they communicate with the outside world: i) a contact dedicated to the power supply (VCC), ii) a contact for resetting the communication (RST), iii) a contact that is used to supply a clock, iv) a contact for powering the card (GND), v) a contact for providing bidirectional half-duplex communication. A sixth contact, which was originally designed for providing higher voltage for programming the card's Electrically Erasable Programmable Read-only Memory Programming (EEPROM), is no longer used. Figure 2.1 depicts the location of the contacts on the card.

The communication with the smart-card can established through various ways. The implementation of the communication messages *i.e.*, how data is transmitted, and the different methods to deal with communication disturbance are defined by the transmission protocols. ISO 7816 defines 15 different transmission protocols, presented as '*T*=' plus a sequential number.

Our work addresses the communication defined by T=0, the most widely used transmission protocol. T=0 was the first smart-card protocol to be standardised, designed in the early years of smart-card technology. The purpose of this protocol is to provide simplicity through minimum memory usage. The communication involves the transmission of bytes and is defined by the Application Protocol Data Unit (APDU).

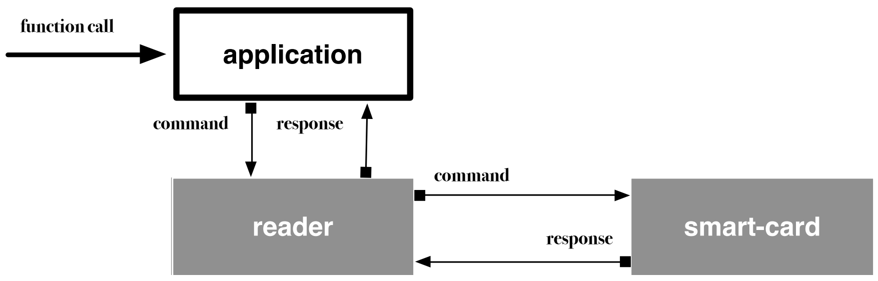


Figure 2.2: Smart-card communication.

APDU messages (defined by ISO 7816) specify the low-level communication between a smart-card and a smart-card reader. Whenever an application receives a function call, it translates that call to low-level communication messages. The messages are transmitted to the card via the reader, and the card responds appropriately. The communication is always initiated by the software via a `command`, in which the smart-card always replies with a `response`. Figure 2.2 depicts that communication. The specification of the communication protocol is defined in a very detailed documentation which comes in 15 parts [Iso.org, 1198, Iso.org, 2007, Iso.org, 2006, Iso.org, 2005f, Iso.org, 2005g],
[Iso.org, 2005h, Iso.org, 2005i, Iso.org, 2005j, Iso.org, 2005k, Iso.org, 2005a], and
[Iso.org, 2005b, Iso.org, 2005c, Iso.org, 2005d, Iso.org, 2005e]. Each part defines a differet aspect of the smart-card setting. In our work we address part 4, 8 and 9 which specify the organisation of the card, security access, the commands for interchange, and the commands for security operations and card management.

Command. Table 2.1 presents the structure of a command. A command consists of a 4-byte compulsory header, the *Cla*, *Ins* and $P1$-$P2$ fields, and an optional variable in length body, the fields *Lc*, *D* and *Le*.

Cla	*Ins*	$P1$-$P2$	*Lc*	*Data Field*	*Le*

Table 2.1: APDU command structure.

In more detail:

- *Cla* indicates the type of the command, *i.e.*, whether it is inter-industry and complies with ISO 7816, or proprietary.

- *Ins* indicates the specific command, eg., SELECT FILE.

- $P1$-$P2$ are the instruction parameters for the command, eg., offset to write into to file to write data.
- *Lc* is the number of bytes of the Data Field.
- *Data Field* contains the data that is sent to the card.
- *Le* is the number of the expected (if any) response bytes.

ISO 7816 specifies the inter-industry command class (*Cla*) and *Ins* codes for all inter-industry commands.

Response. A response, as shown in Table 2.2, consists of an optional variable in length body, the field `Data`, and a compulsory 2-byte trailer, the field `SW1-SW2` which defines the card's status code of the requested operation stated in the incoming command. The length of the response depends on the sent command (Le).

Response Data	$SW1$-$SW2$

Table 2.2: APDU response.

Command-response data exchange. Depending on the initiated operations during the communication, a command-response pair is categorised based on the exchanged data:

- No data is transferred at all. That means that the body of the command message is empty. (Table 2.3).
- Only the response contains data. That means that the body of the command is the field Le. (Table 2.4).
- Only the command transfers data. That means that the body of the command is the field and Lc and the Data Field (Table 2.5).
- Both the command and the response transfer data. That means that the command follows the structure as in Table 2.6.

Cla	*Ins*	$P1$-$P2$

Table 2.3: No data transfer: APDU command structure.

Cla	*Ins*	*P*1-*P*2	*Le*

Table 2.4: Response data: APDU command structure.

Cla	*Ins*	*P*1-*P*2	*Lc*	*Data Field*

Table 2.5: Command data: APDU command structure.

Cla	*Ins*	*P*1-*P*2	*Lc*	*Data Field*	*Le*

Table 2.6: Data in both directions: APDU command structure.

Example of inter-industry and proprietary commands. ISO 7816 specifies the inter-industry command class for the `Cla` field, the allowable values of the `Ins` field and the expected combinations of values for the `P1`, `P2` and `SW1`, `SW2` fields for all inter-industry commands/responses.

Type	`Cla`	`Ins`	`P1`	`P2`	`Lc`	`Data`	`Le`
inter-industry	00	84	00	00	00	00	08
proprietary	80	21	00	00	00	00	08

Table 2.7: Implementations of the get_challenge command.

An APDU implementation is defined according to ISO 7816 and can either be inter-industry, where the command codings are defined by the standards; or proprietary, where the developers define their own command codes. Table 2.7 presents an inter-industry implementation of the get_challenge command and a possible proprietary one. Each byte of the inter-industry command can be decoded, whereas the semantics of the proprietary command is unknown. The inter-industry implementation has its `Cla` field set to 00 as ISO 7816 defines, so, the remaining fields can be decoded. The proprietary one has an unknown `Cla` code, so, it is not possible to determine the semantics of the command using the ISO-based codings. REPROVE aims to infer such unknown semantics.

File organisation and reference methods. According to ISO 7816, a smart-card supports a tree-like file structure:

- *Master File* (MF) is the the root of the file tree.
- *Dedicated Files* (DF) host applications, groups of files, and data objects. A DF

is usually the parent of other files (immediately under this DF). The data that is stored in a DF is a sequence of data objects encoded in TLV (tag, length, value).

- *Elementary Files* (EF) store data and cannot be parents of other files. An EF is categorised to: (i) Internal EFs which store data used by the card for management and control purposes. (ii) Working EFs which store data from the outside world and are not interpreted by the card. The data that is stored in an EF can be referenced as data units, as records or as data objects encoded in TLV depending the supported structure: (i) Transparent structure: the file consists of a single continuous sequence of data units. (ii) Record structure: the file consists of a single continuous sequence of uniquely identified (via numbering) records. (iii) TLV structure: the file consists of a set of data objects.

A file (DF, EF) can be referenced by its unique 2-byte identifier, or by also including the whole path *i.e.*, the identifiers of the parent files. If the file is EF it can also be referenced by a short 5-bit identifier. Short identifiers must be referenced individually as they cannot be included in paths.

Security Status. Security Status is the current state of the card after the completion of:

- Answer to reset and protocol selection, and/or
- A single command or a sequence of commands performing authentication procedures, and/or
- The completion of a security procedure related to the identification of the involved entities, if any. For example
  - by providing the knowledge of a password, eg., `verify` command
  - by providing the knowledge of a key, eg., `get_challenge` command
  - by secure messaging, eg., message authentication.

Security Status is categorised into:

1. *Global Security Status*. It can be modified by the completion of an MF-related authentication procedure *e.g.*, entity authentication by a password or by a key attached to a DF.

2. *File Specific Security*. It can be modified by the completion of a DF-related procedure *e.g.*, entity authentication by a password or by a key attached to a specific DF.

3. *Command Specific Status*. This security status exists only during the execution of a command involving authentication using secure messaging.

Security attributes. ISO 7816 specifies the following security attributes:

1. *Entity Authentication With Password*: The card compares received data from the outside world with internal data.

2. *Entity Authentication With Key*: The entity to be authenticated has to prove the knowledge of a relevant key on an authentication procedure. For example a `get_challenge` command followed by an `external_authenticate` command.

3. *Data Authentication*: Using secret or public internal data, the card checks data received by the outside world. Another way is for the card to check secret internal data and compute a data element (cryptographic checksum or digital signature) and insert it to the data sent to the outside world.

4. *Data Encipherment*: Using secret internal data, the card enciphers a cryptogram received in a data field, or using internal data (secret or public) the card computes a cryptogram and inserts it in a data field, possibly together with other data.

2.3 Security Protocols

A *Security Protocol* is any protocol, abstract or concrete, that performs security-related operations, usually by incorporating cryptographic mechanisms. The main purpose of such protocols is to secure communication in untrusted channels. Security protocols define how the different algorithms can be used in a secure way. Traditionally, security protocols should satisfy the *CIA* triad, *Confidentiality*, *Integrity* and *Availability*:

i) Confidentiality: ensure that the exchanged messages remain secret to unauthorised entities; only the intended recipients should be able to access them.

ii) Integrity: ensure that any unauthorised changes to the exchanged messages can be detected.

iii) Availability: ensure that the system is always available to its users.

It is common that the CIA triad is extended to include the *Authenticity*, *Non-repudiation* and *Privacy* properties:

iv) Authenticity: ensure that the identity of a message's sender is the correct one.

iiv) Non-repudiation: associate actions to a unique individual within a system.

iiiv) Privacy: keep secret an individual's private data and any part of the communication that can be associated with that individual.

In general, a security protocol should protect against attacks. Designing and implementing a security protocol requires profiling the attacker *i.e.*, identify the capabilities, the goals and the strategies of the attacker, with respect to the protocol's specifications. This process, named *threat modelling*, consists of a systematic analysis of the attacker's profile with respect to potential threats to the system. To defend against threats, security protocols incorporate symmetric or asymmetric cryptographic mechanisms. As such, a central aspect of security protocols is the secure distribution of the used keys.

The cryptographic scheme that is used by the protocol defines the need of a secret key exchange. For example, asymmetric algorithms do not require a secret distribution as the encryption is done with public key. The communicating parties circulate their keys publicly and messages are always encrypted with the recipients key. Symmetric schemes though, require that the communicating entities share the same secret for encryption and decryption. Circulating a secret key among communicating entities in an insecure channel rises the *key exchange problem*. In 1976 Whitfield Diffie and Martin Hellman addressed that problem and proposed the so-called *Diffie-Hellman* protocol [Diffie and Hellman, 1976] which is widely used until today. The Diffie-Hellman protocol defines how two entities can establish a secret shared key without revealing any information. The protocol between two entities, A and B proceeds as it follows:

- A selects two random prime values g and p and sends them to B.
- B selects a random secret value a, computes $g^a \mod p$ and sends the result to A.
- A selects a random secret value b, computes $g^b \mod p$ and sends the result to B.
- B computes $(g^b \mod p)^a \mod p$ which will be the value of the key that B possess.
- A computes $(g^a \mod p)^b \mod p$ which will be the value of the key that A possess.

After that session both entities will conclude to the same computation value, since $(g^b \mod p)^a \mod p = g^{ab}$ and $(g^a \mod p)^b \mod p = g^{ba}$.

The protocol on its own however, does not address authentication *i.e.*, there is no way for the involved entities to prove their identities. A solution to that problem is by involving the entities' authentication to the key agreement process. The *Password-authenticated-key-exchange* (PAKE) mechanism utilises an entity's password to the key generation. The principle idea is that the communicating parties share the same password and can all compute the same key based on that password and the exchanged information (as in Diffie-Hellman). This method guarantees that only authorised entities can compute that key, under the assumption that the password is unknown to the adversary. PAKE is widely adopted in commercial applications, as it allows the computation of high entropy keys based on weak human memorable passwords.

According to ISO 7816 smart-cards support similar mechanisms, termed *secure messaging*. The goal of secure messaging is to ensure the authenticity and the confidentiality of the exchanged messages. Secure messaging is achieved by incorporating security mechanisms that involve cryptographic algorithms (*e.g.*, encryption, digital signatures, certificates, MACing *etc.*) and shared keys between the API and the card. Secure messaging is optional; the vendor decides whether it will be enforced by the implementation, and how the API and the card establish the shared keys (*e.g.*, hardcoded from the manufacturer, or via a key exchange protocol).

2.3.1 Attacks

In this section we discuss threats at the protocol layer *i.e.*, the communication between the involved individuals. We do not tackle possible threats to the underlying cryptography nor physical attacks on cryptographic hardware. Our purpose is not to describe all possible attack scenarios; rather it is to provide an introduction to terms that are used throughout this thesis.

A general categorisation of the attacks is into *passive* and *active* attacks. Passive attacks do not interfere with a system; they aim at gaining information by observing its behaviour. Active attacks interfere with the system; they consist of a set of unexpected actions, or actions that originate from an illegitimate source. Attacks can be further categorised based on the attack strategy into *relay*, *man-in-the-middle* and *replay* attacks.

Relay attack. The attack in which the adversary forwards the communication between two entities without them noticing it, is called *relay*. For example, consider two communicating entities A and B and the adversary M. As presented in Figure 2.3, the adversary captures the transmitted messages m_B and m_A, and then forwards them unchanged to the original recipient respectively. A relay attack can also be active, in which the adversary initiates the communication between the entities.

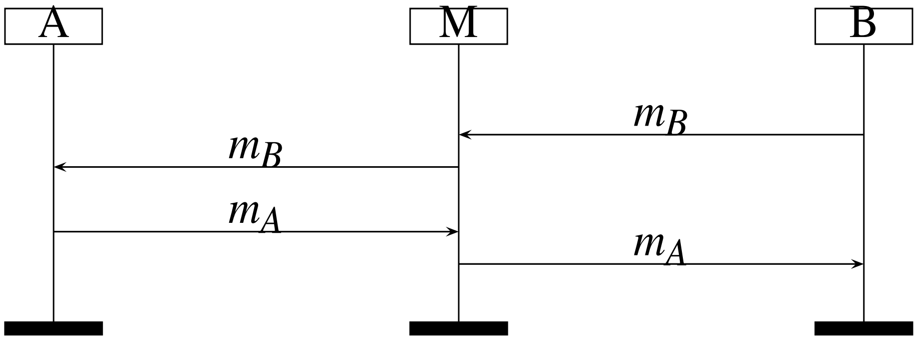


Figure 2.3: Example: the relay attack.

Man-in-the-Middle attack. When the adversary interferes with the communication between two entities, the attack is called *man-in-the-middle* (MitM), and can be passive or active.

In passive attacks, also called *eavesdropping*, the adversary only observes the communication (but does not relay it). Such attacks usually aim at getting access to the transmitted data (possibly private) and/or deducing the protocol's properties. Figure 2.4 demonstrates an example, in which the adversary M eavesdrops the communication between the entities A and B.

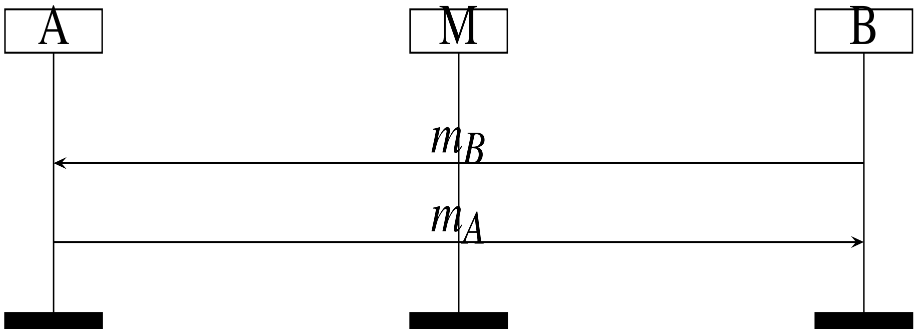


Figure 2.4: Example: passive man-in-the-middle attack.

In active MitM attacks the adversary impersonates an involved entity and tampers with the communication, by altering, blocking or injecting messages. The goal of the adversary is to alter the outcome of the communication in a way which will provide him with access to private data or allow him to control one of the involved entities. For example, as Figure 2.5 demonstrates, when B sends the message m_B to A, the adversary changes it to $m_{B'}$ and forwards it to A. The same tactic can be applied to A's messages.

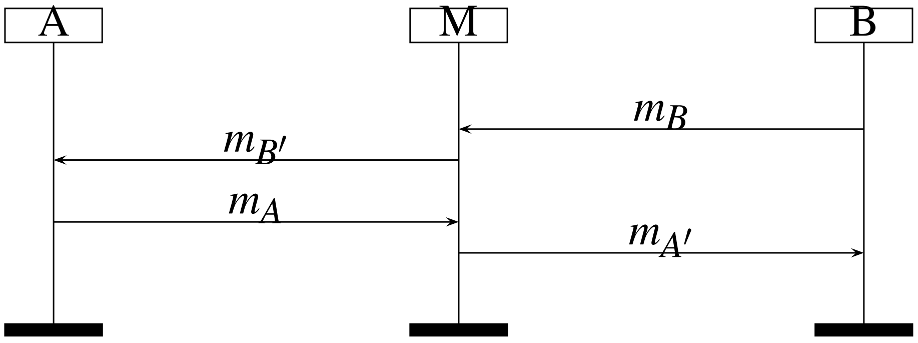


Figure 2.5: Example: active man-in-the-middle attack.

Replay attack. An attack in which the adversary resends exchanged messages from an earlier round between legitimate entities is called *replay*. The goal of this attack is to impersonate an entity and/or to cause a protocol error which will allow access to private data or unauthorised control of an entity. Figure 2.6 demonstrates a replay attack in which the adversary replays the message m_B that B had previously sent to A.

Replay attacks can occur during: i) the same protocol round, *i.e.*, while the communication is still active between two entities, ii) a new protocol round, *i.e.*, during the execution of the protocol the adversary replays messages that appeared in an earlier round, or initiates a new round by replaying the messages of a previous one, iii) a different protocol, *i.e.*, the adversary replays parts of a protocol into a different one.

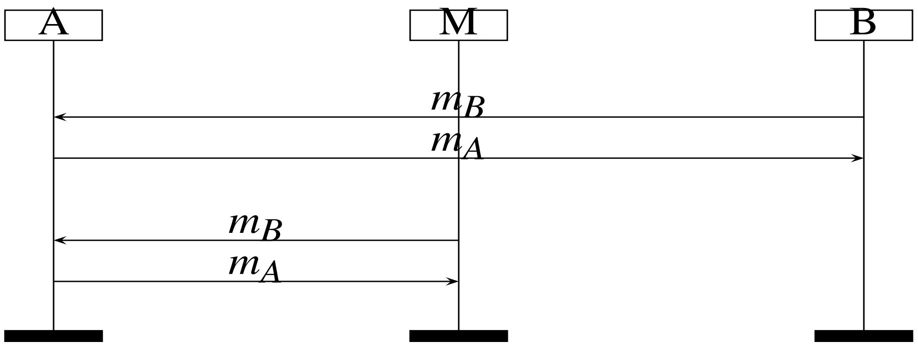


Figure 2.6: Example: the replay attack.

2.3.2 Smart-Card Attacks

Different approaches for attacking smart-cards exist, depending on whether physical access to the card is required. The attacks can be categorised into physical and non-physical.

Physical. The attacks that by using special equipment target the physical characteristics of the card are called *physical*. They are categorised into *invasive*, *semi-invasive* and *non-invasive*. *Invasive* attacks require the removal of the card's microprocessor and involve physically tampering with it. Usually this type of attack is time-consuming and require expensive equipment. For example, the process may involve the reverse-engineering of secure blocks to extract read only data from the card's ROM (Read Only Memory) [Nohl et al., 2008]. Another example would be placing probes on the communication buses to read the exchanged data with an oscilloscope [Anderson and Kuhn, 1996].

Semi-invasive attacks involve exposing the surface of the card's chip but do not require its modification to compromise its security. Semi-invasive attacks can use lasers, electromagnetic fields and tools for ionizing radiation such as UV light, X-rays to observe or inject faults *e.g.*, [Quisquater and Samyde, 2001, Gandolfi et al., 2001, Skorobogatov, 2005]. *Non-invasive* attacks target the power consumption of the card and involve observing the information that can be leaked during the computation of a function or injecting faults *e.g.*, [Anderson and Kuhn, 1996, Kocher et al., 1999] and [Mangard et al., 2008, Bar-El et al., 2004].

Non-physical. The attacks that do not target the physical aspects of a smart-card are called *non-physical* or *logical attacks*, and compromise the card's security by exploiting possible implementation flaws at the software level. Logical attacks are less general than the physical ones, as they exploit the specific application of the card *i.e.*, the high-level protocol incorporated by that card, and/or the card's underlying implementation protocol. Different attacks and methodologies have been proposed depending on the capabilities of the card. We categorise the attacks into the ones that target the software coded on the card and the ones that target the low-level communication.

Attacks on the card's software are applicable to multi-application cards, *i.e.*, Java cards. A common practice is to exploit flaws of the loaded applet or the underlying platform (Java). The attacks are achieved by sending malicious inputs or by loading a malicious applet to the card, *e.g.*, [Mostowski and Poll, 2008]
[Iguchi-Cartigny and Lanet, 2010, Barbu et al., 2011], or by overflowing the APDU

buffer *e.g.*, [Barbu et al., 2012].

Attacks on the low-level communication require getting control of the layer that enables it. This can be achieved by accessing the Personal Computer-Smart Card (PC/SC) layer. PC/SC is a midleware library that manages the smart-card communication. There exist dedicated dedicated malware softwares that provide that control in Linux machines *e.g.*, [Ludovic, 2016]. In our work the experiments were conducted on a Windows machine. The operating system has a dedicated library, winscard.dll, for managing the low-level communication. To sniff the exchanged messages we used the WinSCard APDU View Utility software [1] which interacts with winscard.dll and allows sniffing and relaying the communication. Apart from software-based solutions, there also exist hardware specifically designed to allow eavesdropping and tampering with the communication, *e.g.*, [Choudary, 2010, De Koning Gans and de Ruiter, 2012].

The attacks that target the low-level communication follow the same principles as in the security protocol attacks (relay, man-in-the-middle, replay). The attack strategy depends on the application of the card and the adversary's goal. For example, payment transactions have been proved to be vulnerable to man-in-the-middle attacks in which the adversary tampers with the card's response to get an incorrect PIN verified *e.g.*, [Murdoch et al., 2010a, Anderson et al., 2006]. Another example is exploiting insecure implementations of the communication, which allow unauthenticated sessions to be initiated, and tampering with the data that is stored in the card. For example, a set of PKCS#11 smart-cards have been proved vulnerable to such attacks [Bozzato et al., 2016].

[1] Available at www.fernandes.org/apduview/ –last time accessed at June 2016.

Part I

PKCS#11 Smart-cards

Chapter 3

PKCS#11: Introduction

Analysing, attacking and fixing cryptographic standards used by smart-cards, such as RSA PKCS#11, is an active area. As defined in PKCS#11 [RSA Security INC, 2004], cryptography is only one aspect of security and the token is only one component in a system; one must consider the environment the token operates in as well. Smart-cards supposedly offer a tamper-resistant environment for protecting sensitive data, but should also be designed so that this data remains secure. This is delegated to the communication protocols, under the assumption that these protocols are secure. Proprietary implementations create the sense of security as they hide the card's code. A smart-card operates as a black-box: only access to the card's code may reveal the semantics of the communication protocol and its internal operations. We propose reverse-engineering the smart-card communication protocol, with respect to PKCS#11, to determine the security of that implementation. We present REPROVE, which stands for Reverse Engineering of PROtocols for VErification: an automated tool based on first-order logic, that infers the semantics of the communication, the on-card operations and their interconnection with PKCS#11. REPROVE is implementation- and function-independent, as it deals with both inter-industry and proprietary implementations and does not require access to the card's code.

An alternative to REPROVE's automated reasoning is to manually reverse-engineer the trace. This is not straightforward and is far from a quick exercise. It requires access to the card's library and its internal calls, whereas REPROVE does not. If one tries to *guess* the meaning of the trace, without access to the card, then, given the combinatorial nature of the problem, one will need to test a considerably large number of combinations (*e.g.*, in some of the cards we tested there are more than 540×86^8 possible combinations—see also Chapter 5.4) which will require a long time to decode.

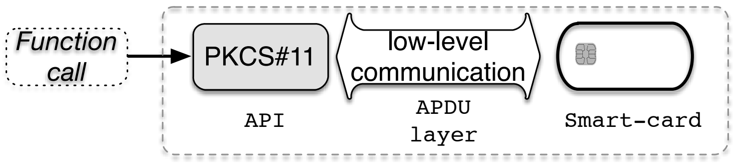


Figure 3.1: API and smart-card interaction when a PKCS#11 function is called.

REPROVE does this in a matter of milliseconds. An example of the required effort for manual reverse-engineering is presented in [Bond et al., 2004] where the researchers spent approximately 7 months to deduce the implementation the *Chrysalis – ITS Luna CA*[3] PKCS#11 token.

PKCS#11 smart-cards intend to provide additional security to a system by offerring a secure and tamper-resistant environment for executing cryptographic functions and managing the corresponding secret data. The whole setting is considered to be secure under the assumption of a trusted path between the user and the card: by verifying his credentials (PIN) the user proves his legitimacy, whilst the card is always considered to be trusted, perform the requested operations and ensure that secret data is never exposed. The cards are used in addition to a dedicated Application Programming Interface (API) which allows the user to access the card's operations. Any API call (*i.e.*, calling a specific cryptographic function) initiates a low-level communication that manifests as a communication trace of API requests to the smart-card, as shown in Figure 3.1.

REPROVE reverse-engineers the low-level implementation of the cryptographic protocol by automatically aligning the byte-wise decomposition of the communication trace to the expected PKCS#11 calls for specific types of functionality. This process is helpful in multiple ways:

- It provides the means to test smart-card implementations and discover their security vulnerabilities.
- In the absence of detected vulnerabilities, it provides empirical evidence for the security of the implementation.
- It can be used by the developers of smart-card technologies to test their implementations.
- It can be used by the clients themselves, to test whether their card is vulnerable to attack and, therefore, fraud.

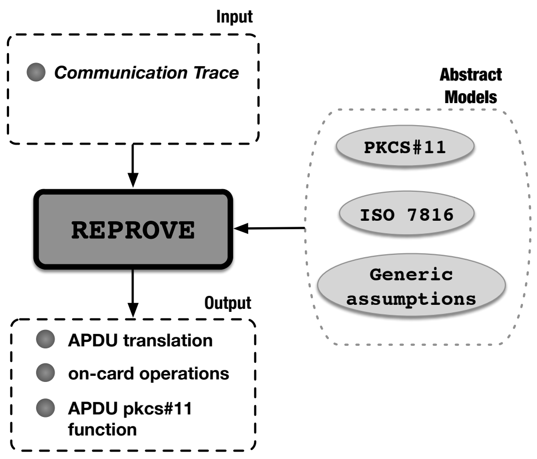


Figure 3.2: High-level overview of our reverse-engineering technique.

A security token, such as a smart-card, implements all the cryptographic operations internally. The token stores objects (*e.g.*, data and certificates) that can be accessed via session handles, and performs cryptographic functions. PKCS#11 is the most widely used cryptographic standard of functions like signing, encryption, decryption, *etc*. API-related attacks were first discovered in [D. Longley, 1992], followed by the exposure of the vulnerability to attacks of PKCS#11 [Bortolozzo et al., 2010, Clulow, 2003]. Formally analysing security APIs and reasoning about attacks has been revisited [Youn et al., 2005b, Steel and Bundy, 2005, Courant and Monin, 2006a, Delaune et al., 2008] through approaches like model checking, theorem proving, customized decision procedures, or reverse-engineering for verification *e.g.*, [Youn et al., 2005a, Courant and Monin, 2006b, Cortier et al., 2007] and [Delaune et al., 2008, Bortolozzo et al., 2010]. However, security analysis has mostly focused on the PKCS#11 itself. There has been less attention given to the implementations connected to the standard, such as the low-level communication between the on-card and the off-card applications, defined by the Application Data Protocol Unit (APDU).

The basic principles of the APDU, *e.g.*, the structure and the contents of the exchanged messages, the available inter-industry commands *etc*, are specified by the ISO 7816 standard. Precisely following the standard is not compulsory. Many smart-card manufacturers deviate from the standard under the assumption that a proprietary APDU implementation is more secure. REPROVE reverse-engineers the APDU implementation and deduces the card's functionalities, regardless of whether it is inter-industry, proprietary or a mixture of both.

A high-level description of REPROVE is shown in Figure 3.2. The card communicates with the reader and this communication generates a trace that we reverse-engineer. The analysis module accepts as parameters the trace and abstract models of the cryptographic protocols; and outputs how the card performs specific cryptographic functions. It models the low-level communication in first-order logic, and uses reasoning and inference over plug-in knowledge bases, which consist of APDU abstractions based on ISO 7816 and PKCS#11, to automatically reverse-engineer the model. Its algorithm parses a communication trace and uses these abstractions to draw conclusions about the semantics of the various elements of the trace, narrow-down their possible implementations and infer the card's actually executed operations.

To the best of our knowledge, this is the first work for modelling the APDU layer and formally reverse-engineering it by mapping the low-level communication to the on-card operations and to the PKCS#11 standard. The abstract models of the background knowledge do not hard-code the implementation. Instead, they offer a generic framework to automatically capture different implementations. Specific implementations are mapped to these abstractions by reasoning about the exact meaning of the input trace. Our novelty stems from not requiring access to the card's software and dealing with both inter-industry and proprietary implementations in a single setting.

Contributions and Roadmap.

The main contributions of this work are:

- Chapter 4 gives an overview of the PKCS#11 standard and its known vulnerabilities. A basic overview of Prolog, REPROVE's implementation language, is also provided. Finally, the chapter discusses the related work in the field. As this is the first work to address the analysis of proprietary APDU implementations, and also the interconnection between the PKCS#11 standard we discuss works that lye in similar lines as ours.

- Chapter 5 provides the implementation details of REPROVE. Section 5.2 presents the modeling of the APDU layer and its interconnection to PKCS#11. We only model a subset of the PKCS#11 functions which we will later use to analyse the security properties of the tested smart-cards. The set consists of functions that deal with sensitive and private operations. REPROVE, however, is function independent as it is possible to plug in different models. Computing all potential proprietary implementations and testing them for correctness is practically infeasible, as it is a combinatorial problem. Instead, we produce a model that is based on decomposing the various functionalities of the API into finer-grained sub-functionalities and analyse how the commands of the standard can be used to implement these functionalities. We present the reverse-engineering algorithm to automatically analyse a trace of commands and group them according to their intended functionality as this has been captured by our model. Section 5.4 evaluates the accuracy of REPROVE, after reverse-engineering seven commercially available smart-cards for nine cryptographic functions.

- Chapter 6 provides the security analysis of the reverse-engineered smart-cards. Section 6.1 introduces a threat model for PKCS#11 smart-cards. We present and categorise the potential threat scenarios of PKCS#11 smart-cards. We identify commonalities between the various attacks and we narrow-down the attack search space to a set of vulnerable parts. We use this set to to conduct a security analysis on the reverse-engineered smart-cards. The analysis is presented in Section 6.2.

  Automatically verifying such implementations is not a trivial task. The implementations being diverse makes it difficult to define comprehensive formal models. We present a semi-formal method for discovering vulnerabilities and implementation flaws based on state-machines. We show how REPROVE can cooperate with other systems toward that goal. Section 6.3 presents an ecosystem in which REPROVE and smartCardLearner [Ruiter, 2015] coordinate to extract meaningful state-machines of the smart-cards. Finally we showcase practical APDU attacks in Section 6.4.

Chapter 4

Background

The first part of this thesis investigates the low-level implementations of PKCS#11 smart-cards. Section 4.1 provides the necessary background on the PKCS#11 standard. As the standard itself is described in a long documentation [RSA Laboratories, 2009, OASIS, 2015], we only focus on the aspects that our work addresses. Also, in this section we survey the literature of the PKCS#11 vulnerabilities, focusing on the ones we address. Section 4.2 provides some preliminaries on REPROVE's underlying implementation language and the representation that we will use throughout this document. Automated analysis of APDU implementations is a novel idea, with REPROVE being the first of its kind, to the best of our knowledge. However, there do exist other works related to ours, which we discuss in Section 4.3.

4.1 The PKCS#11 Standard

Security APIs are intended to provide access to sensitive resources in a secure way. The design of such APIs is critical, as they have to ensure the secure creation, deletion, importing and exporting of a key from a device. Also, they are responsible for permitting the use of these keys for encryption, decryption, signing and authentication so that even if a device is exposed to malicious software the keys remain secure. The RSA PKCS#11, also known as the Public-Key Cryptography Standard, specifies an ANSI C API, called *Cryptoki*, for hardware devices that can perform cryptographic functions and store cryptographic-related and encrypted data. It aims to 'sand-box' an application and isolate it from the details of the underlying cryptographic device.

Cryptoki is intended as an interface between applications and a hardware device that is managed by a single user. It does not have the means of distinguishing multiple

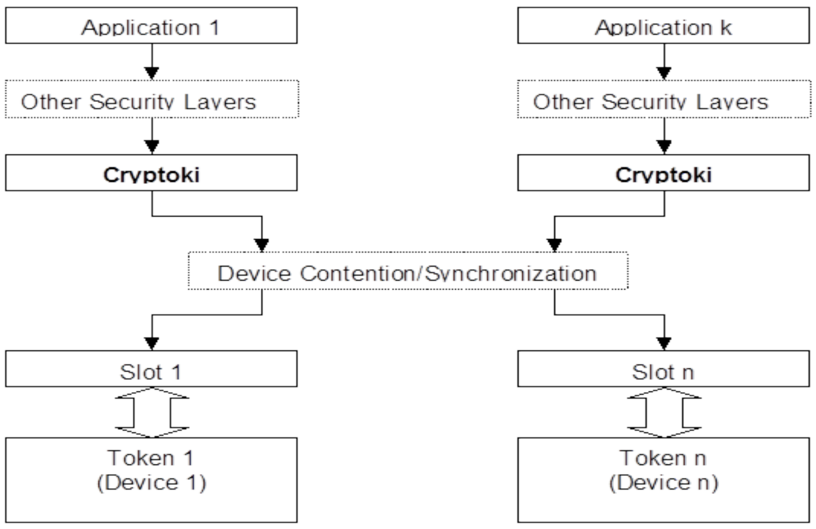


Figure 4.1: The general model of Cryptoki.

users. As such, it focuses on keys and the corresponding public certificates of a single user. The general model of the layers of a system that utilises Cryptoki, as illustrated in [RSA Security INC, 2004] is presented in Figure 4.1. The model begins with one or more applications that require the execution of cryptographic operations, and ends with the cryptographic tokens which are responsible for executing these operations. Cryptoki aims at providing to the applications a programming interface which abstracts the details of the cryptographic tokens. Cryptoki may provide a general interface for more than one cryptographic device which belong to the same system. Each device connects to Cryptoki via a slot (physical reader) and can perform a set of cryptographic operations which are required by a higher level application.

When an application connects to a security token it authenticates itself and initiates a session which is either public or private, defining the kind of objects the application can access and the types of operations that it can perform on them. Each session is assigned a unique id by the Cryptoki, the session handle, which aims at preventing replaying of the the same session. The application can then access the token's objects *e.g.*, keys and certificates.

4.1.1 Objects and Attributes

PKCS#11 defines three classes of objects:

1. Data: general purpose data defined by an application.

2. Cryptographic keys: depending on the algorithm used a key object can be a

Attribute	**Definition**
`CKA_VALUE`	the value of the object
`CKA_PUBLIC_EXPONENT`	the public exponent and the modulus of RSA asymmetric keys
`CKA_PRIVATE_EXPONENT`	the private exponent for RSA asymmetric keys
`CKA_LABEL`	the label of the object
`CKA_SENSITIVE`	defines the object as sensitive; the object cannot be wrapped
`CKA_EXTRACTABLE`	defines that the object can be extracted from the device
`CKA_PRIVATE`	defines that the object requires the user authentication
`CKA_SIGN`	defines that the object can be used for digital signatures
`CKA_ENCRYPT`	defines that the object can be used for encryption
`CKA_DECRYPT`	defines that the object can be used for decryption
`CKA_WRAP`	defines that the object can be used for the wrapping of another object
`CKA_UNWRAP`	defines that the object can be used for unwrapping another object

Table 4.1: A sample of the `CKA_` attributes as defined in PKCS#11.

secret key, a public key or a private key.

3. Certificates: public-key certificates.

Objects are also categorised depending 1. their availability outside the device *i.e.*, whether their value is accessible by the software, 2. the period of their validity, *e.g.*, session keys, 3. their access requirements, *i.e.*, whether authentication is required to access them, and 4. the purpose of the object *i.e.*, the operation that this object can be used on. These characteristics are specified via attributes (defined by the `CKA_` class). An example of attributes specified by the standard is presented in Table 4.1. An attribute can either be assigned a string, *e.g.*, `CKA_LABEL`, or a boolean value, *e.g.*, `CKA_SENSITIVE`. An object may be referenced by its handle *i.e.*, a pointer to that object.

Attributes also serve as protection mechanisms of the objects; depending on the class of the object, PKCS#11 defines a set of rules of how attributes are set, for example, which attributes must be true if an object is a private key, when an attribute cannot be changed to FALSE if initially was TRUE *etc.*. Namely, the set of rules that we will discuss during this thesis is presented in Table 4.2. These are the attributes that protect secret and private keys, and changing their value is strictly forbidden by the standard.

Attribute	Rule
`CKA_SENSITIVE`	once set to TRUE cannot be set to FALSE; the object cannot be extracted
`CKA_PRIVATE`	once set to TRUE cannot be set to FALSE; limits the object use to the user
`CKA_EXTRACTABLE`	once set to FALSE it cannot be set to TRUE allows the object to be wrapped
`CKA_SENSITIVE`	if TRUE then `CKA_EXTRACTABLE` must be FALSE

Table 4.2: A sample of the `CKA_` attribute rules as defined in PKCS#11.

4.1.2 Functions

PKCS#11 provides a set of functions for managing the hardware device, session management, object management, encryption, decryption, message digesting, signing and MACing, verifying signatures and MACs, random number generation, parallel function management, callback and dual function cryptographic functions. Not all smartcards execute the entire function set defined by the standard. However, there is an overlap between the functions that manage keys and perform operations over sensitive data. Thus for our work we tested the following subset of PKCS#11 functions:

- `C_login` authenticates a user. A successful call can initiate a private session and provide access to the token's private objects. The function takes as inputs the session handle, the type of the user (user, or a privileged user termed a security officer), the location of the user's PIN and the length of the PIN.
- `C_generateKey` is called to generate a secret key or a set of domain parameters. It takes as inputs the session handle, the location of the generation mechanism, the location of the template for the new key's attributes, the number of attributes in the template and the location of the handle of the new key.
- `C_sign` signs data, with the signature being an appendix to the data. Its inputs are a session handle, the location of the data, the location of the signature and the length of the signature.
- `C_findObjectsInit` is called to initiate a search for token and session objects that match an input template with attribute values. It takes as inputs the session handle, the location of the template and the number of attributes in the template.
- `C_findObjects` is called after `C_findObjectsInit` and obtains the handles of the objects that match the given template. It takes as inputs the session handle,

the maximum number of the returned handles, the location of the additional object handles and the location of the actual number of the returned handles.

- `C_getAttributeValue` is called to obtain the value of one or more attributes of an object. It takes as inputs the session handle, the object's handle, the location of a template with the attribute values to be obtained and the number of the template's attributes.

- `C_setAttributeValue` is called to modify the value of one or more attributes of an object. It takes as inputs the session handle, the objects' handle, the location of the template with the attributes, the number of the attributes to change and the new values of the attributes.

- `C_wrapKey` is called to encrypt a private or a secret key. It takes as inputs the session handle, the location of the wrapping mechanism, the handle of the wrapping key, the handle of the key to be wrapped, the location of the wrapped key and the length of the wrapped key.

- `C_encrypt` is called to encrypt data. It takes as inputs the session handle, the data to be encrypted, the location of the encrypted data and the length of the encrypted data.

- `C_unwrapKey` is called to decrypt a wrapped key and creates a new private key or a secret key object. It takes as inputs the session handle, the location of the unwrapping mechanism, the handle of the unwrapping key, the wrapped key, the length of the wrapped key, the location of the new key, the location of the template of the new key, the number of the attributes in the template and the location of the handle of the new key.

4.1.3 PKCS#11 Attacks

The security of cryptographic APIs has been extensively studied [Delaune et al., 2008, Delaune et al., 2010, Fröschle and Sommer, 2011, Centenaro et al., 2012, Künnemann, 2015, Scerri and Stanley-Oakes, 2016] based on Clulow's pioneer *key-separation* attacks [Clulow, 2003]. Clulow suggested that the lack of enforcement of separation between encryption, authentication and wrapping keys can give conflicting roles to a key. For example, the same key should never be used for wrapping and decrypting: given two keys k_1 and k_2 an attacker can determine the value of k_1 by wrapping it with

k_2 and then decrypting the resulting ciphertext with k_2. A key can be given conflicting roles via its attributes. As such, a secure PKCS#11 implementation should impose some restrictions on how the attributes of a key can be set. According to [Fröschle and Sommer, 2011] the possible PKCS#11 attacks are categorised based on the adversary's goals, to:

- Key conjuring attacks: the adversary obtains a freshly generated key by calling the `C_generateKey` or `C_generateKeyPair` function.

- Trojan Key attacks: the adversary injects his own key k_a into the token. This can be achieved by:

  i. calling the `C_createObject` function, or ii. knowing a symmetric key k with `CKA_ENCRYPT` and `CKA_UNWRAP` set, the attacker wraps k_a with k, and then unwraps $\{k_a\}_k$ with k, or iii. given a key k_u with `CKA_UNWRAP` set, and a key k_{enc} with `CKA_ENCRYPT` set, the attacker first encrypts k_a with k_{enc} and then unwraps $\{k_a\}k_{enc}$ with k_u.

- Role tampering attacks: the adversary is able to change the attributes of a key by adding a new attribute to the key which conflicts with the existing ones.

- Downgrade attacks: decrease the security of a key by changing its attributes, *e.g.*, set `CKA_SENSITIVE` to FALSE.

- Upgrade attacks: the adversary increases the security of a key to cause denial of service.

The work described in this thesis lies within the aforementioned attacks. In particularly the PKCS#11 vulnerabilities that we consider are:

- *Faulty Key-management.* Misuse of the key attributes, violations on the key-generation specifications and lack of restrictions lead to faulty key management. Such behaviour opens back-doors for a series of attacks.

  i) *Insecure Key Generation.* Sensitive keys must always be generated inside the hardware device and never be exposed in plaintext. API-side key generation is considered as a bad practice as the software itself does not offer a tamper-resistant environment. Such behaviour defeats the purpose of cryptographic tokens.

ii) *Conflicting Attribute Allocation.* Each key k is specified by an attribute template $\{att_1, att_2, .., att_n\}$ where each att_i specifies the allowed usage of that key. For example, for a sensitive key *sk* `CKA_SENSITIVE` is always set, whereas `CKA_EXTRACTABLE` must never be set. Such pairs of attributes are called *conflicting attributes* as they cannot be set `true` at the same time. If a configuration allows setting both `CKA_SENSITIVE` AND `CKA_EXTRACTABLE` to true, key *sk* can be exposed.

iii) *Key usage not compliant with the attributes.* Depending on the key's purpose different attributes are set. A faulty key management does not take into consideration the restrictions defined by the attribute leading to incorrect usage of the key. An example would be wrapping the key *sk* whereas `CKA_NEVER_EXTRACTABLE` is set.

- *Unauthorised Access to the Token.* The implementation allows an unauthorised entity to have access to the token and consequently to PKCS#11 functions.

- *Replay of Sessions.* Each initiated session between the token and the API should be assigned a handle. Session handles are unique identifiers whose purpose is to prevent replaying and ensure authorised access to the cryptographic functions. One of the most commonly addressed PKCS#11 violations is the lack of session handles. Although session handles is not the only measure against replay attacks, when combining with trivial authentication mechanisms *e.g.*, a simple PIN verification, such attacks cannot be avoided. Examples of real-world applications that have been affected by this misconfiguration are the RBS Worldpay [Cryptosense, 2014] and the DigiNotar [DigiNotar, 2014] incidents.

Moreover, we address vulnerabilities with regards to the data that is handled by cryptographic functions. In particular data which is not considered as sensitive by the standard, but it is private for the user. For example, the message to be encrypted, the decryption result, the message to be signed.

4.2 Prolog

To be able to verify if an APDU implementation fulfils the necessary security requirements, only identifying the semantics of the exchanged commands is not sufficient. The problem boils down to i) inferring the semantics of the command, and possible relations and dependencies between them; ii) extracting the operations that are performed, with respect to the exchanged commands, iii) identifying the set of operations that implement a specific PKCS#11 function. This creates the need of identifying abstractions and patterns for the communication and then refining them to the appropriate commands. During this process, the models need to evolve through a constant rewriting process. Logic fulfils these requirements. We consider the problem of APDU analysis to be a semantic interpretation and knowledge reasoning problem. Our approach is to model the APDU protocol in first-order logic as it is expressive enough to model the protocol rules, without restricting them. REPROVE is written on Prolog, as it bridges the gap between completeness and readability.

Prolog is a logic programming language based on First-Order Logic. Being a declarative language, it considers programs as theories in formal logic and computations as deductions in that logic space. Prolog is based on Horn clauses, with a built-in unification algorithm. Any program written in Prolog is a set of models that express facts and rules with regards to a given domain. Both rules and facts are declaratively written in the form of clauses. A rule is of the form:
$H : -B_1, B_2, .., B_n.$
which describes that if B_1 and B_2, and ..., and B_n are `TRUE`, then H is `TRUE`. H forms the head and $B_1, B_2, .., B_n$ forms the body of the rule. Facts are clauses with empty bodies and are of the form: *p*/*n*, where *p* defines the name of the predicate and *n* its arity.

Prolog's execution mechanism is resolution based and has a built-in backward chaining inference engine which is used to derive conclusions from a given knowledge base. Given an input query, it tries to find a resolution refutation of the negated query. If a negation is not found, then it follows that the query is a logical consequence of the program followed by the provided variable binding. One of the most powerful features that Prolog provides for the APDU analysis is backtracking, also known as depth-first search.

The operations a smart-card executes and consequently the command-response pairs that are responsible for initiating them, are quite often dependant to each other. The effectiveness of a command depends on the satisfiability of a precondition set from the previous ones. As such, inferring the command itself heavily depends on the previous ones, and the meaning of that one will affect the next ones, leading to a tree with different interpretations of the same trace. Back-tracking provides a good solution to that problem by starting the search from the root and exhaustively proceeding to the last node of the tree. As our knowledge modelling is based on logic, Prolog is a good fit since it provides an automated mechanism for exhaustive search space exploration while itself providing a logic-based interface.

4.3 Related Work

Protocol reverse-engineering. Protocol reverse-engineering is a related area to our work, but works up to this date do not satisfy the requirements of our project, which addresses a lower-level of abstraction. For example, Polyglot [Caballero et al., 2007] is a system for automatically extracting the format of the protocol messages. The system uses dynamic binary analysis during the protocol execution. In more detail, the system monitors the binary program that implements the protocol while it processes inputs. Their intuition is that the way a program processes its inputs may reveal information about the incoming messages. Polyglot focuses only on extracting the format of the messages rather than the semantics, and requires access to the binary program of the protocol.

Similarly, Discoverer [Cui et al., 2007] is a system that reverse-engineer the protocol message format. The system infers protocol idioms commonly seen in message formats of many application-level protocols, from the application's network trace. Discoverer groups the messages with similar sequences of text or binary tokens and then recursively clusters them and aligns the sequences until it produces a more detailed description of the message formats.

Another similar approach is Prospex [Comparetti et al., 2009], which infers the protocol format and the corresponding state machines. The system focuses on the *behaviour* of the messages: it identifies and clusters messages based on their structure and on the impact that each message has on the server. Moreover, the system extracts the protocol's state-machine based on the sequences of messages the protocol permits Prospex uses dynamic taint analysis for monitoring an application while it processes incoming messages to extract the *behaviour* of the messages only. A similar work is presented in [Cho et al., 2010] in which the users suggest inferring protocol states machines of the messages by incorporating the user's feedback on the protocol abstractions.

The aforementioned systems deal with plaintext protocol messages only. ReFormat [Wang et al., 2009] is a system for extracting the format of encrypted messages. The system first identifies the location of the decrypted message by analysing the run-time buffers, and then reveals the associated message structure.

All these systems focus on deriving the protocol specifications rather than the semantics of the messages. Although the smart-card low-level communication in other application areas (*e.g.*, EMV) follows a particular protocol, in PKCS#11 it is more random. The communication between different cards varies significantly and bares little or none similarities. Thus, extracting specification with respect to an implemented protocol cannot succeed in our project. Moreover, such techniques i) require access to the API, and/or

ii) assume known message semantics, and/or

iii) derive only the protocol message format without its semantics.

What is central to our work is to make no assumptions about a specific communication protocol. The only knowledge we can use is what is publicly available, *i.e.*, the inter-industry commands of ISO 7816. Using that knowledge, our goal is to infer proprietary semantics and deduce the card's operations.

Analysis of APDU implementations. Automated reverse-engineering of the APDU layer is a novel concept and has not been studied before. However, there exist related works that address the importance of understanding that layer and note that it is possible that the card might not have the expected behaviour. A system that aims at inferring a card's behaviour with respect to the APDU layer is SmartCardLearner [Ruiter, 2015]. The authors implemented an automated system for inferring state-machines of a card's APDU implementation. SmartCardLearner takes as input a set of APDU commands that are recognisable by the card and by systematically sending all possible commands combinations to the card, SmartCardLearner extracts a state-machine that shows the exact sequences of commands that are permitted. The output is useful to understand the underlying implementation as well as to identify possible flaws. However, the state-machines are useful only if the semantics of the commands is available.

The work presented in [Bozzato et al., 2016] is the closest to ours. The authors manually analysed the APDU implementation of a set of PKCS#11 smart-cards. Their technique was through manual inspection of the API and library calls, API tests and debugging of the library. REPROVE was able to reverse-engineer a larger collection of cards than the one presented in [Bozzato et al., 2016]. The authors provided us with the results of the manual analysis which we used to evaluate REPROVE's performance.[1]

[1]The authors of [Bozzato et al., 2016] are also co-authors of the REPROVE published work [Gkaniatsou et al., 2015], as they provided us with the implementation details of the smart-cards we tested on REPROVE. These details were used to evaluate REPROVE's accuracy. Our work was pub-

Namely, the cards are: Aladdin eToken PRO, Athena ASEkey, RSA SecurID, Safesite Classic TPC V1 and Siemens CardOS V4.3b. Both the work presented in this thesis and in [Bozzato et al., 2016] address the importance of analysing the APDU layer. A manual analysis though, requires a deep understanding of the APDU layer, a thorough study of the extensive documentation of ISO 7816, access to the API and performing systematic tests. Instead, our solution consists of a fully automated system which produces results in milliseconds and does not require any effort from the user.

In [Bozzato et al., 2016] the authors also showed that it is possible to tamper with a key's `CKA_SIGN` attribute. In particular, they set the attribute from `FALSE` to `TRUE` which allowed them to use that key for signatures. For completeness we reproduce that attack to a smart-card that was not studied in [Bozzato et al., 2016]. Moreover, we present a novel attack that has not been previously discussed (more details can be found in Section 6.4).

lished in 2015. In 2016 the aforementioned authors also published that manual analysis.

Chapter 5

REPROVE: Automatically Reverse-engineering the Application Protocol Data Layer

REPROVE aims to provide insight into the smart-card communication protocol, by inferring models of the characteristics of the implementation. To do so, our reverse-engineering approach focuses on both the semantics and the abstractions of the communication to allow for higher-level reasoning, as opposed to focussing only on the literals exchanged. In that way, by running REPROVE through a trace, the user can fully understand what is happening without having to go through any extensive documentation that defines that layer. To provide such a deeper understanding, our reverse-engineering approach boils down to:

1. Developing a formal model for the communication layer that will be used as generic background knowledge for REPROVE's reverse engineering algorithm.
2. Incorporating rules for the logical requirements specified by the standard. These requirements are extracted by carefully studying the documentation of ISO 7816 to infer semantic links, dependencies and restrictions that, while not formally defined by the protocol, are necessary for any communication based on it.
3. Creating links between the low-level input, the operations and their abstractions.
4. Enabling REPROVE to work at a higher abstraction layer to be able to work for a larger class of cards in a uniform way and, in the end, better translate the communication trace to on-card operations.

The background knowledge is modelled in first-order logic and consists of abstract models which need to be instantiated according to the input trace. These models are based on ISO 7816 and define: i) the communication language, ii) the properties, restrictions and requirements of the communication, iii) implementations of the on-card operations in terms of communication, iv) possible implementations of specific PKCS#11 functions. Such models do not hard-code the implementation of the card; they present abstractions of different functionalities that are then refined according to the input trace. REPROVE will translate the input trace based on the background knowledge and provide an abstraction of it in terms of on-card operations by applying the restrictions and specifications defined on its rules. More formally, REPROVE applies the transformation function $g(f(x))$ with $f : T^n \rightarrow I^n$ and $g : I^n \rightarrow O^m$, where T^n is an input trace of n commands, I^n is a set of n inter-industry commands (1-1 mappings) and O^m is a set of m on-card operations.

5.1 ISO 7816 Logical Requirements

Transmission protocol. ISO 7816 defines different transmission protocols, however all PKCS#11 compliant smart-cards only follow the T=0 transmission protocol: the communication is always initiated by the interface device, which is the only participant that can send requests, while the card can only reply to the reader's requests. Thus, we only consider commands whose origin is the interface device, responses that originate from the smart-card. Moreover, we assume that i) the card always responds to a command, ii) the card's response is always related with the incoming command *e.g.*, if a command requests some binary data, then the response will only contain the requested data.

***Ins* codes.** As defined in ISO 7816, we consider the values of the instruction codes, *i.e.*, *Ins* fields, to be unique. However, according to ISO 7816, the $P1$ and $P2$ fields may indicate different interpretations of the same *Ins* code. We categorise the commands that have multiple interpretations; during the analysis of a command, if at least one mapping falls in this category, then the assumption of uniqueness is dropped. This does not affect the analysis.

Files and data structures. Commands indicate particular data structures. However, commands that address different data structures may occur within the same trace. Based on the above specification we restrict a file[1] from having data stored in multiple

[1]ISO 7816 specifies a single data structure per file rule. The card may handle multiple structures in

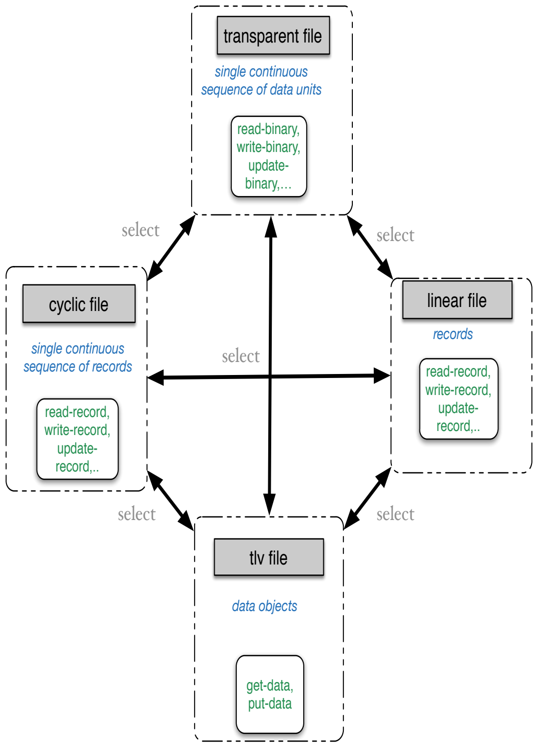

transparent file
single continuous sequence of data units
read-binary, write-binary, update-binary,...
select
select
cyclic file
single continuous sequence of records
read-record, write-record, update-record,..
select
linear file
records
read-record, write-record, update-record,..
select
select
tlv file
data objects
get-data, put-data

Figure 5.1: The file and data structures as defined by ISO 7816, and the ways to reference them.

structures and we identify the valid file structures: i) transparent, ii) linear, iii) cyclic, iv) tlv, whereas, each file structure corresponds to a specific data structure: i) data object, ii) data unit, iii) record. We identified the interconnection between the file, the data structures and the way to reference them, as shown in Figure 5.1, and we consider it possible to reference different structures if and only if a different file is selected. With respect to these properties, a group of commands that indicate different data structures in the same file is considered as invalid. We use the above restrictions to narrow-down the mapping candidates for each proprietary command.

Command pairs and card operations. A pair defines dependent commands; if a command appears within the trace, then its paired command should also be. Such pairs are defined based on the card's operations. This allows us to eliminate invalid mappings from the beginning of the analysis. Furthermore, we assume that the card will execute at least one operation.

different files.

5.2 Modelling the Application Protocol Data Unit Layer

REPROVE's intent, in addition to inferring the semantics of the communication, is to deduce the card's functionalities and infer mappings between different PKCS#11 functions and the APDU layer. As such, we perceive different abstractions of that layer. A bottom-up view of the communication, with respect to REPROVE's modelling is the following:

1. Communication: the semantics of exchange communication messages.

2. On-card operation: an operation executed on the card which has been initiated by the communication messages. The parameters of the operation are also provided by such messages.

3. PKCS#11 function: a function defined in PKCS#11 which is executed by a set of on-card operations.

In Figure 5.2 we show a high-level description of our modelling approach, with respect to the APDU abstractions. Each individual card operation (*functionality*) of the card, is decomposed into a sequence of steps (*sub-functionalities*). Each step is then implemented as a sequence of APDU commands: proprietary, inter-industry, or a mix of the two. The APDU commands are further characterised depending on their data exchange properties (shown, for example, as 'YY' in the figure to indicate a command that both sends and receives data) and their role within the sub-functionality in question (that we have termed *core*, *additional*, or *dummy* and we will refine further in the following sections). The same command may have different data exchange properties and different roles depending on the sub-functionality, *e.g.*, command$_a$ and command$_x$ in Figure 5.2. The following sections provide details of how each abstraction is modelled.

5.2.1 Commands

REPROVE's background knowledge consists of 26 commands, as presented in Table 5.1.

In the models, an APDU command is represented as a predicate of arity seven: `command(Cla,Ins,P1,P2,Lc,D,Le)` where the variables *Cla*, *Ins*, *P1*, *P2*, *Lc*, *D*,

select	get_data	update_record
read_binary	read_record	write_binary
update_binary	erase_record	write_record
activate_file	put_data	generate_asymmetricKeyPair
get_response	perform_security_operation	delete_file
append_record	create_file	deactivate_file
manage_security_environment	get_challenge	erase_binary
verify	external_authenticate	mutual_authenticate
general_authenticate	internal_authenticate	

Table 5.1: REPROVE's background knowledge: set of commands.

command	**Ins**	**#interpretations**
create_file	1	1
activate_file	1	8
delete_file	1	8
deactivate_file	1	8
select	1	8
read_binary	2	3
write_binary	2	4
update_binary	2	10
erase_binary	2	20
get_data	2	8
put_data	2	10
read_record	2	2
erase_record	1	1
append_record	1	2
write_record	1	6
update_record	2	3
get_response	1	1
get_challenge	1	1
generate_asymmetricKeyPair	2	16
perform_security_operation	1	15
manage_security_environment	1	18
verify	2	6
external_authenticate	1	3
mutual_authenticate	1	3
general_authenticate	2	18
internal_authenticate	1	4
total		187

Table 5.2: Summary of the different command representations and interpretations.

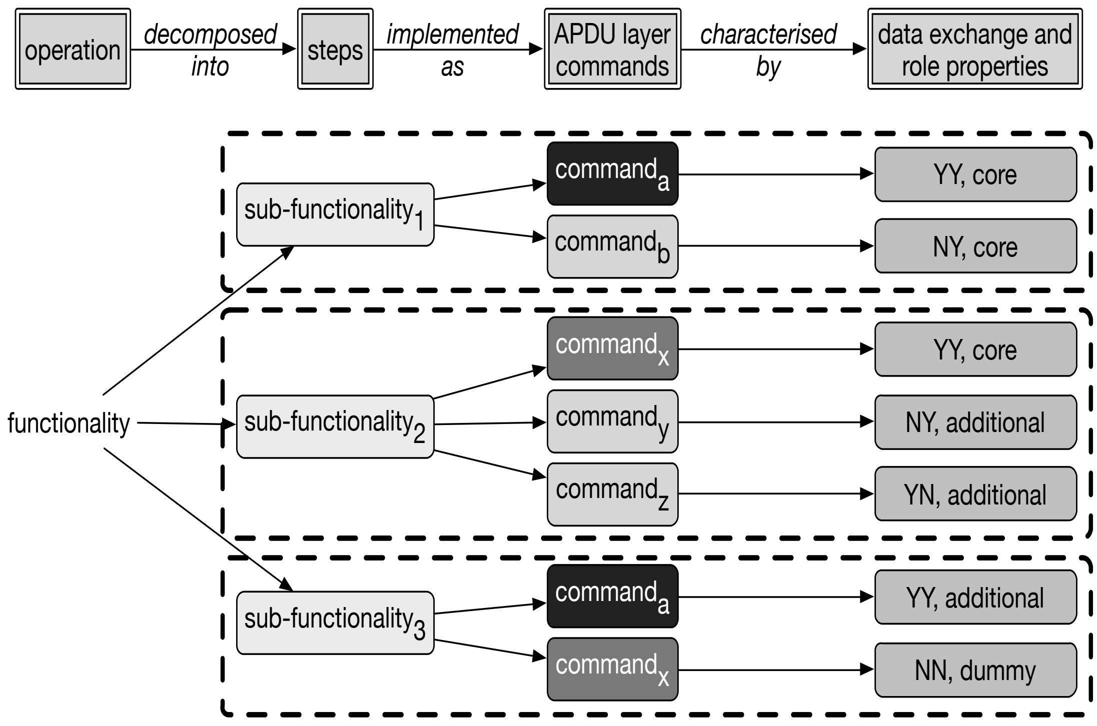


Figure 5.2: A single operation represents a specific functionality and it is modeled as a sequence of sub-functionalities. Each sub-functionality is further implemented as a sequence of commands. Commands are characterised by their data exchange properties and role within some particular sub-functionality.

Le are instantiated according to the specifications defined by ISO 7816. A command is considered valid if it falls in one of the following categories: i) any inter-industry command; and ii) any proprietary command that can be mapped to an inter-industry command if and only if this inter-industry command has not occurred within the same implementation[2], and has all its preconditions satisfied.

According to ISO 7816, a command may be referenced by more than one *Ins* code. The interpretation of a command (*i.e.*, the operation requested and the conditions that are true if the command succeeds) depends on the instantiations of the $P1$, $P2$ and D fields. Additionally, a command may have multiple interpretations. For example, the `generate_asymmetricKeyPair` command can be referenced either by the 46 *Ins* code or by the 47 code. $P1$ $P2$ and D specify the different parameters for the key generation operation *e.g.*, the algorithm that will be used for the generation, possible information on the generated key pair, access to an existing public key,

[2]We consider each representation to be unique: the exact same instantiations cannot occur at both an inter-industry and a proprietary command, within the same trace.

whether the card should return the public key, the format of the returned data *etc.* Depending on the command's instantiations, there are 32 different interpretations for the `generate_asymmetricKeyPair` command[3].

Table 5.2 presents the number of the different *Ins* codes and the different interpretations of each command. The background knowledge consists of 187 rules that specify the conditions under which a different command interpretation occurs, as defined by ISO 7816. Each rule defines the action that the command is responsible for, and the conditions for that action. For example, the rule:

```
command(00,a4,08,04,Lc,D):-
                 isa(D, ef),
             select(file, D).
```

states that if the elementary file `D` is selected, then the command fields are instantiated as `Cla=00`, `Ins=a4`, `P1=08`, and `P2=04`.

5.2.1.1 Command Categories

A command is categorised based on: i) its data exchange properties; and ii) the card operations.

Categorization according to data exchange properties. We have defined rules that assign each command, depending on the exchanged data, to one of the following categories:

(i) $\mathsf{command}_{nn}(Cla, Ins, P1, P2, Lc, D, Le)$: no data is sent, no data is expected,

(ii) $\mathsf{command}_{ny}(Cla, Ins, P1, P2, Lc, D, Le)$: no data is sent, data is expected,

(iii) $\mathsf{command}_{yy}(Cla, Ins, P1, P2, Lc, D, Le)$: data is sent, data is expected,

(iv) $\mathsf{command}_{yn}(Cla, Ins, P1, P2, Lc, D, Le)$: data is sent, no data is expected.

Variables *Lc*, *D* and *Le* define the category of a command. For instance, the rule:

```
command_nn(Cla, Ins, P1, P2, Lc, D, Le) :−
  command(Cla, Ins, P1, P2, 00, 00, null).
```

describes that if $Lc = \texttt{00}$, $D = \texttt{00}$ and $Le = \texttt{null}$[4] then the command does not send or retrieve any data.

[3]In our models we include only RSA objects, as our experiments use that algorithm. The models however can be extended to include other algorithms as well.

[4]The value `null` indicates absence of a field.

Categorization according to card operations. For each operation of the card we have categorised the commands into:

(*i*) *Core*: the basic commands that perform the operation, *e.g.*, to create a new file `create_file` is a core command.

(*ii*) *Additional*: the commands that add extra properties to the operation, but they do not change its meaning; the same operation can be implemented without them. For example, to create a file `select` is an additional command as it merely adds information to file creation (*e.g.*, selecting a path to create the file into) but file creation can proceed without it.

(*iii*) *Dummy*: the commands that neither send nor expect any data. They usually just query, or check, the communication with the card, for example, a `verify` command, when it does not send nor expect any data to/from the card. Such commands may occur at any time during the communication and they do not change the output of the reverse-engineering.

The role of a command varies depending on the operation.

5.2.1.2 Command preconditions

The preconditions of each command define: i) The types of previously issued commands. For example, a `read_binary` command is applicable only if the previous commands have selected an elementary file *i.e.*, `select(file,D)` and `isa(D,ef)` must be true. ii) The data types and file structures of the previous commands should match, for example, a `read_binary` command can be applicable after a `read_record` command only if they address a different file.

5.2.2 Card Operations

The implementation of an operation varies depending on the data object, the file structure, the algorithms and the protocols that a smart-card supports. We propose a hierarchy of abstractions that capture the distinct implementations of the same operation. We introduce the umbrella term *functionality*, which represents a high-level view of an operation, to define the *purpose* of the operation rather than the implementation details. A *functionality* is decomposed into *sub-functionalities* which represent the specific steps needed to implement different instantiations of an operation.

Functionalities. *Functionalities* represent the generic operations that a smart-card may execute *e.g.*, authentication, data-retrieval, data-storing *etc.*, based on the purpose for which the smart-card is used[5]. With respect to ISO 7816 and the analysed PKCS#11 functions, we have defined the following *functionalities*:

(*i*) *read_ data*: operations for data-retrieval.
(*ii*) *store_ data*: operations that modify the contents of the card *e.g.*, write, update, create
(*iii*) *authentication*: operations for authentication.
(*iv*) *generate_ key*: operations for generating a key/key-pair.
(*v*) *sign*: digital signature operations.
(*vi*) *verify*: operations for verifying a digital signature.

A *functionality* models the set of *sub-functionalities* needed for this operation to be executed, with regards to the underlying implementation mechanism. The models are defined by *core* and *additional sub-functionalities* which capture the various implementation steps. The *core sub-functionalities* define the necessary steps for that operation. The *additional sub-functionalities* define extra steps that change the characteristics of that operation, but the operation can be implemented without their execution. In the background knowledge, a *functionality* is represented by the predicate `functionality(F, S, C, Sen)`, in which `F` defines the name of the operation, `S` the set of *sub-functionalities* that correspond to that operation, `C` the set of the core *sub-functionalities* that must be satisfiable and `Sen` the set of sensitive *sub-functionalities*, *i.e.*, the operations that deal with sensitive data, with regards to the analysed PKCS#11 function.

Each *functionality* may be defined by more than one model; each models describes a different implementation mechanism. For example, *store_ data* is described by the following models:

(*i*) `functionality(store_data(L, D, S), [read_data_sub(F,Le, RD), file_create(L, D)],[file_create(Location, D)], Sen)`: First the data `RD` is retrieved from `F`. Then a new file `D` at the location `L` is created. The core sub-functionality is *file_ _create* while *read_ data_ sub* is an additional.[6]
(*ii*) `functionality(store_data(L, D, S),[read_data_sub(L,Le, RD), data_write(L, D)],[data_write(L, D)], Sen )`: First the data `RD` is retrieved from the location `L`. Then at the same location, `L` some new data `D` is

[5]Different purpose may indicate different capabilities. For example, PKCS#11 smart-cards can verify a signature while Bitcoin smart-cards cannot.

[6]The variable `Sen` will be instantiated by the reverse-engineering algorithm, according to the analysed PKCS#11 function.

written. [6]

Sub-functionalities. A *sub-functionality* represents the set of the *actions i.e.*, the command outcomes, required for the execution of a specific operation. Given an abstract operation, we have defined the following set of of *sub-functionalities*, according to the the different implementation mechanisms a smart-card can support (as stated in ISO 7816):

(*i*) *data_read_sub*: the mechanisms for requesting data from the smart-card.

(*ii*) *data_write*: the mechanisms for storing/altering data to the card.

(*iii*) *file_create*: the mechanisms for creating a new file/directory in the card.

(*iv*) *file_activate*: the mechanisms for activating a file/path of the card.

(*v*) *file_delete*: the mechanisms for deleting a file/path of the card.

(*vi*) *data_delete*: the mechanisms for deleting data that is stored in the card.

(*vii*) *secret_verify*: the mechanisms that request the verification of a secret provided by the API.

(*viii*) *mutual_authenticate*: the mechanisms that implement a mutual authentication between the smart-card and the API based on a shared key.

(*ix*) *external_authenticate*: the mechanisms that authenticate the API to the smart-card, based on a shared key.

(*x*) *internal_authenticate*: the mechanisms that authenticate the smart-card to the API.

(*xi*) *generate*: the mechanisms that request the generation of a key/key-pair.

(*xii*) *perform_digital_signature*: the mechanisms for requesting from the smart-card to perform a digital signature.

(*xiii*) *verify_digital_signature*: the mechanisms for requesting from the smart-card to verify a digital signature.

(*xiv*) *hash*: the mechanisms for requesting from the smart-card to compute a cryptographic hash.

(*xv*) *encrypt*: the mechanisms for providing some data to the smart-card and requesting its encryption.

(*xvi*) *decrypt*: the mechanisms for providing some encrypted data to the smart-card and requesting its decryption.

Each *sub-functionality* is represented by the predicate `sub-functionality(S, C, Cc, Sen)`, in which `S` defines the name of the sub-functionality, `C` the set of the commands that satisfy that operation, `Cc` the set of the core *commands i.e.*, the commands that are necessary to implement that particular operation, and `Sen` the set of sensitive *commands*, *i.e.*, the commands that deal with sensitive data, with regards to the analysed PKCS#11 function.

A *sub-functionality* may have more than one model, each describing the different underlying implementations. REPROVE contains in total 43 such models. Each model consists of a set of *core* commands, which represent the preconditions of a *sub-functionality*. If the set of the *core* commands is not `TRUE`, then the model *sub-functionality* cannot be `TRUE`. For example, *data_read_sub* is defined by the following models:

(*i*) `subfunctionality(read_data_sub(DF, Le, RD), [[isa(DF, offset), isa(RD, record), retrieve_data(Le, DF, RD)]], [retrieve_data(Le, DataType, RD)], Sen)`: The operation retrieves the `RD` records from the DF offset. [6]

(*ii*) `subfunctionality(read_data_sub(card, Le, RD), [[isa(F,ef), select(file, F), isa(RD, binary), retrieve_data(Le, binary, RD)]], [retrieve_data(Le, binary, RD)]), Sen)`: The operation selects the elementary file `F` and retrieves the `RD` binary data. [6]

Each model describes a different way of retrieving data, according to the different data structure, file structure and ways of accessing it. In all models the core commands are the one responsible for retrieving data; the additional commands are the ones responsible for defining the access methods and the data and file characteristics.

In the *sub-functionality* models, instead of including the representation, commands are expressed by what we define their *interpretation*: what is actually achieved by sending this command. For example, instead of `command(00,a4,08,04,04,50154400)` the command is represented as `(isa(50154400, ef), select(file,50154400))` which defines the action of selecting the elementary file `50154400`.

5.2.3 PKCS#11 Models

PKCS#11 models represent assumptions on how each PKCS#11 function is expected to be implemented. These models aim to capture an abstraction of the expected on-card operations so they do not impose an implementation, but merely act as a flexible guide of the implemented functionality. The models are expressed in terms of functionalities and are represented by the predicate `pkcs(N,F)`, where `N` is the name of the PKCS#11 function and `F` the set of *functionalities* that are responsible for carrying out that function.

Each function may be described by more than one model *i.e.*, a different set of *functionalities* , depending on the assumptions made on the implementation. For example, `C_logIn` is described by the following models:

- `pkcs(log_in, [authentication(A,B, S)])`: an authentication operation is implemented.
- `pkcs(log_in, [read_data(Location, File, RD,S), authentication(A,B,C)])`: first some authentication related data is retrieved *e.g.*, permitted incorrect PIN attempts, and then the authentication operations is implemented.

The predicate `authentication/3` corresponds to one of the following mechanisms: i) authentication with a PIN: the card compares received data from the outside world with internal data; ii) authentication with a key: an entity to be authenticated has to prove the knowledge of a relevant key through the challenge-response procedure; iii) data authentication: using internal data, secret or public, the card checks data received by the outside world. Another way is for the card to check secret internal data and compute a data element (cryptographic checksum or digital signature) and insert it to the data sent to the outside world; iv) data encipherment: using secret internal data, the card enciphers a cryptogram received in a data field, or using internal data (secret or public) the card computes a cryptogram and inserts it into a data field, possibly together with other data.

5.3 REPROVE Reverse-Engineering Algorithm

REPROVE's reverse-engineering algorithm consists of three steps, each addressing a different abstraction of the implementation: (i) the APDU trace semantics, (ii) the on-card operations that are executed during communication, and (iii) the APDU implementation given a PKCS#11 function.

Step 1: Semantics of the APDU Trace. Given an input trace T^n of n commands, we generate a tree in which each path from root to leaf $T_i^{n\prime}$ is a semantic mapping of the trace such that $T^n \mapsto T_i^{n\prime}$. As the exchange of the command-response pairs is sequential so is the analysis of the commands, which implies that the semantics of an unknown command heavily depends on the previous commands. Each unknown command is categorised and all corresponding mappings M are identified, which are

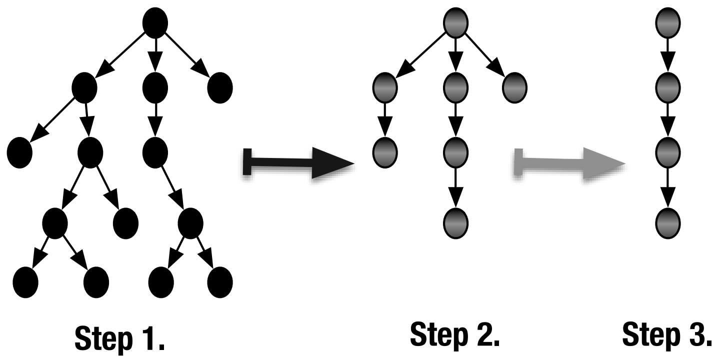


Figure 5.3: Reducing the search space.

then narrowed-down to a set P' based on precondition satisfiability. For each mapping $m \in P'$, the commands analysed so far are grouped, and sets that fully or partially satisfy[7] any sub-functionality are considered valid. The outcome of this process is a set of valid[8] mappings M'' of each unknown command such that $M'' \subseteq P' \subseteq M$, and the set P which consists of different interpretations of T. More formally, Step 1 performs the transformation $f : f(T^n) = P^n$ where $\forall T_i^{n'} \in P^n : T^n \mapsto T_i^{n'}$.

Step 2: On-Card Operations. At this stage, given P^n from the previous step, the commands at each $T_i^{n'} \in P^n$ are grouped in all possible combinations. Each group is checked to determine whether there exist any sub-functionality(ies) that satisfy its preconditions. The outcome of this process is a set S^l of sub-functionalities such that $\forall S^l{}_k \in S^l \exists T_i^{n'} \in P^n : T_i^{n'} \mapsto S^l{}_k$. Then all sub-functionalities in S^l are grouped and the set of valid functionalities O^m is identified. The sub-functionalities that do not satisfy O^m are discarded along with the corresponding trace mappings. The overall step can be represented as a function y: $y(P^n) = O^m$ with $S^{l'} \mapsto O^m$, $S^{l'} \subseteq S^l$, and $P^{n'} \mapsto S^{l'}$, $P^{n'} \subseteq P^n$.

Step 3: PKCS#11 Function. Here, the set of functionalities O^m from Step 2 is mapped to the background models of specific PKCS#11 functions, resulting in an interpretation of the communication in terms of the standard. The outcome is the APDU mapping to PKCS#11, the set of card operations that are executed during the communication, $O^{m'} \subseteq O^m$, and the APDU traces $T_i^{n'} \in T^{n'}$ that satisfy them.

Figure 5.3 shows how we restrict the search space: grey arrow indicates narrowing-down and black arrow indicates mapping; each path of a black tree is an individual

[7]Given a sub-functionality, there exists at least one core command that satisfies its preconditions.
[8]Valid here indicates that neither the ISO, nor any background model is violated.

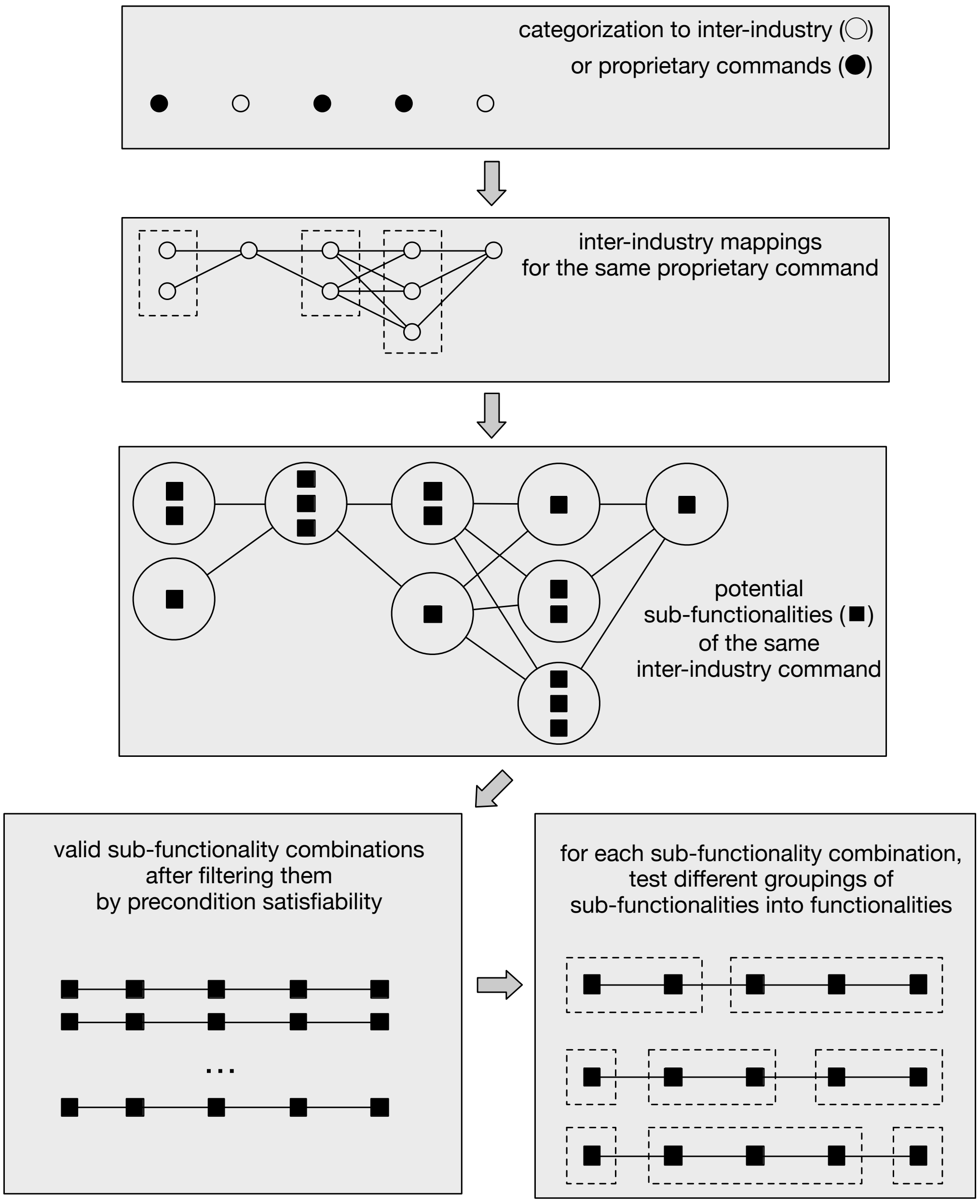

categorization to inter-industry (○)
or proprietary commands (●)
inter-industry mappings
for the same proprietary command
potential
sub-functionalities (■)
of the same
inter-industry command
valid sub-functionality combinations
after filtering them
by precondition satisfiability
for each sub-functionality combination,
test different groupings of
sub-functionalities into functionalities

Figure 5.4: The transformations of the APDU trace during the reverse-engineering process.

mapping of the same APDU trace. The nodes appearing at the same depth represent different mappings of the same command; each path of a grey tree represents a sequence of executed card operations (sub-/functionalities). Step 1 generates a tree of all the command mappings, where each path is a different trace mapping. The mapping of a command affects the mapping of the subsequent command; thus, not all paths have a valid depth *i.e.*, the same as the number of commands in the trace. In Step 2 the command paths of valid length are mapped to functionality paths (on-card operations). Finally, Step 3 discards functionality paths that do not match with the PKCS#11 models.

In each reverse-engineering step the low-level input (commands) evolves to abstract models (card operations). A schematic description of the transformations of the commands during the reverse-engineering process is presented in Figure 5.4. The trace itself goes through a sequence of transformations: from commands, to inter-industry mappings, to potential sub-functionalities, to groups of sub-functionalities into higher-level functionalities. If REPROVE is successful in providing a sequence of functionalities that describe a PKCS#11 function, then the trace is effectively reverse-engineered. This is considered as a drawback of the card as it exposes its implementation and potentially opens attack vectors (as we present in the following chapter).

Reverse-Engineering Algorithm. The overall reverse-engineering process for a trace of commands is shown in Algorithm 1[9]. The input to the algorithm is a list $\mathscr{T}$ of commands representing the communication trace, and the output is a list P of potential mappings of $\mathscr{T}$ (each mapping is a list itself) and a list O of card functionalities. The list P is initialised to [[]] which indicates that the first mapping is the empty one. Each command $c \in \mathscr{T}$ is then analysed and depending on its value of Cla it is classified as proprietary or inter-industry. In the former case (lines 3 to 5) the values of its L_c, D and L_e parameters are checked to categorize its data exchange properties and obtain a list $\mathscr{M}$ of potential mappings. From $\mathscr{M}$ we only keep the valid mappings (lines 5 to 12) and store them in P. The valid mappings are identified based on precondition and sub-functionality satisfiability (lines 6 to 9): for each potential mapping to an inter-industry command, we check that the preconditions of the inter-industry command are met by computing the union of the postconditions[10] of all commands that precede it. If the preconditions of an inter-industry command are not met, the erroneous mapping is removed from $\mathscr{M}$ and the analysis continues to the next candidate mapping; else, we iterate over the analysed trace so far, and look at the categorization of commands based on their role. Using this role, we group commands into different combinations that may form potential sub-functionalities. If such a grouping exists, the mapping is stored in P. If c is an inter-industry command, there is only one such mapping n, so $\mathscr{M}$ is a singleton list. We search for satisfiable sub-functionalities by

[9]In Algorithm 1 we only show the conceptual reverse-engineering process to aid the presentation. The actual implementation of the algorithm is in Prolog and leverages the analysis algorithm of the language.

[10]Postconditions represent what is `TRUE` given the particular interpretation of a command

this command and store the command in *P* (lines 13 to 17). At this point *P* consists of different mappings of the trace. Then, *P* is further narrowed-down based on the sub-functionality and functionality models (lines 18 to 25). For each different mapping of the trace, the commands are grouped into sub-functionalities which are then further grouped into higher-level functionalities that are added to *O*, all in the context of our models. If no such grouping is found for a candidate trace, the trace is removed from *P*. If a grouping is found, its constituent mappings are annotated accordingly to denote this. The final step of the algorithm is to further narrow-down *P* by matching the resulting functionalities in *O* with the PKCS#11 models. In the end, *P* will contain zero or more traces of candidate mappings. If *P* is empty, our reverse-engineering has failed to produce a mapping. If there is only one trace in *P* we say that the mapping is unique. If there is more than one candidate trace we need further experiments, for example a manual analysis, to eliminate the false traces.

5.4 REPROVE Evaluation

5.4.1 Evaluation Setting

We have evaluated REPROVE using seven commercially available smart-cards. Each smart-card had its own closed-source API implementation, provided by the manufacturer. We were not able to test the same PKCS#11 functions for all cards. This is because in some cases the cryptographic function was executed library-side instead of token-side (*i.e.*, outside the card instead of on-card), which is violation of the standard as it allows for sensitive data, *e.g.*, keys, to be transmitted outside of the token. In other cases, the API did not allow the execution of the function at all *e.g.*, the `C_wrapKey` function.

Our purpose was to assess REPROVE along the following dimensions:

- Functional success: the system infers at least one model. If REPROVE is unable to infer a model then, there are two cases: (*i*) the system has failed, or (*ii*) the communication is encrypted. The latter case is not REPROVE's failure as it merely acts as a verification that the implementation is secure.
- Quality of the results: the output captures at least a high-level view of the implementation *i.e.*, the implemented operations. REPROVE can produce more than one output models. We consider the following outcomes to be of high quality: (*i*) a unique model which matches exactly both with the low- and the high-level

Algorithm 1: The reverse-engineering process for a trace of commands.

```
input  : List $\mathscr{T}$ of commands to be analysed
output: Potential mappings and operation models $P$ for $\mathscr{T}$

1  $P = [[]]$;    $O = [[]]$;
2  foreach $c(Cla, Ins, P_1, P_2, L_c, D, L_e) \in \mathscr{T}$ do
3      if Ins indicates c is proprietary then
4          use $l_c, d, l_e$ to extract data exchange properties $d$;
5          $\mathscr{M}$ = list of APDU commands $c$ maps to based on $d$;
6          foreach $m \in \mathscr{M}$ do
7              $Z = \{z \mid (k \text{ precedes } m \text{ in } p) \wedge (z \in \text{postconditions}(k))\}$;
8              if preconditions of m are not satisfied by Z then
9                  remove $m$ and move on to the next;
10             foreach $p \in P$ do
11                 if a grouping of p to sub-functionalities can be found then
12                     $s = p \oplus (c \mapsto m)$;    $P = P \oplus s$
13     $n$ = inter-industry command $c$ maps to;    $\mathscr{M} = [n]$;
14     annotate each command with its sub-functionality;
15     annotate sub-functionalities with functionalities;
16     $O = O \oplus functionalities$;
17     $s = p \oplus (c \mapsto n)$;    $P = P \oplus s$;
18 foreach $p \in P$ do
19     foreach $(c \mapsto m) \in p$, potential sub-functionality of m do
20         group sub-functionalities into functionalities;
21         if no such grouping can be found then remove $p$ from $P$;
22         else
23             annotate each command with its sub-functionality;
24             annotate command groups with functionalities;
25             $O = O \oplus functionalities$;
26     foreach $f \in O$ do
27         if $f \notin$ PKCS#11 models then remove $f$ from $O$;    remove $p$ from $P$ ;
28 return $P, O$;
```

views of the implementation, *i.e.*, the exchanged commands and the on-card executed operations, and (*ii*) two or more models that exactly match the high-level view of the implementation, *i.e.*, on-card executed operations.

To address these aspects we used the standard precision and recall metrics, as defined by:

$$\text{precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$
$$\text{recall} = \frac{\text{True Postives}}{\text{True Positives} + \text{False Negatives}}$$

where (*i*) True Positive: the outcome model suggests the correct on-card operations and the exact meaning of the APDU trace, (*ii*) False Positive: the outcome model suggests the correct on-card operations, and a partially correct meaning of the APDU trace (the APDU semantics does not exactly match with the actual implementation), (*iii*) False Negative: the outcome model suggest incorrect on-card operations and an incorrect meaning of the APDU trace.

For each smart-card we used the sniffed APDU trace as the input to REPROVE. The trace was produced when each PKCS#11 function was called. We were aware of the implementation of each smart-card from the beginning but we treated them as unknowns during the reverse-engineering. To evaluate the quality of the results we compared REPROVE's output with the actual implementation. The smart-cards that we tested are: Aladdin eToken Pro by Gemalto (ex Safenet), ASEKey USB by Athena, CardOS V4.3 by Atos [11], CardOS V4.4 and CardOS V5 offered by Atos, SecurID offered by RSA, TCP ISV1 offered by Safesite.

5.4.2 Results

Number of inferred models. REPROVE performed well on all cards: it inferred at least one model for the exchanged commands, one model for the on-card operations and one model for the analysed cryptographic function. In most cases the inferred model was unique and matched exactly with the actual implementation of the card. The results are presented in Table 5.3. For Aladdin eToken Pro, and Athena ASEKey USB in the case of `C_logIn` and for Safesite TCP ISV1 in the case of `C_sign` REPROVE inferred the correct on-card operations but suggested two different implementation models, hence the 0.5 precision value. In all cases the correct model of the implementation existed within the suggested ones.

Narrowing-down the search space. The reverse-engineering of proprietary APDUs is a combinatorial problem and the solution time grows exponentially with the size of the APDU trace. REPROVE uses search to advance towards the proof, and inference to block and exclude directions from the search. During the analysis, the search space is

[11](The CardOS V4.3 used to be manufactured by Siemens.)

Smart-card	Function	Precision	Recall
Aladdin eToken Pro	`C_logIn`	0.5	1
	`C_wrapKey`	1	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_getAttributeValue`	1	1
	`C_generateKey`	1	1
	`C_getAttribute`	1	1
	`C_encrypt`	1	1
Athena ASEKey USB	`C_logIn`	0.5	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_generateKey`	1	1
	`C_setAttributeValue`	1	1
	`C_encrypt`	1	1
Atos CardOS V4.3	`C_logIn`	1	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_getAttribute`	1	1
	`C_setAttribuyeValue`	1	1
Atos CardOS V4.4	`C_logIn`	1	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_getAttribute`	1	1
	`C_setAttribuyeValue`	1	1
Atos CardOS V5	`C_logIn`	1	1
	`C_sign`	1	1
	`C_findObjects`	1	1
	`C_getAttribute`	1	1
	`C_setAttribuyeValue`	1	1
RSA SecurID	`C_logIn`	1	1
	`C_findObjects`	1	1
	`C_getAttributeValue`	1	1
	`C_sign`	1	1
Safesite TCP ISV1	`C_logIn`	1	1
	`C_sign`	0.5	1
	`C_setAttributeValue`	1	1

Table 5.3: RSA PKCS#11 reverse-engineering evaluation results.

Smart-card	**Function**	**#B.CC**	**#R.CC**	**#R.SFC**	**#R.FC**	**#R.Model**
Aladdin eToken Pro	`C_logIn`	13932	24	11	3	2
	`C_wrapKey`	20	4	1	1	1
	`C_sign`	20	8	1	1	1
	`C_findObjects`	20	3	1	1	1
	`C_generateKey`	86	9	2	1	1
	`C_getAttribute`	400	6	1	1	1
	`C_encrypt`	200	4	1	1	1
Athena ASEKey USB	`C_logIn`	32000	12	4	2	2
	`C_sign`	20	24	1	1	1
	`C_findObjects`	400	3	1	1	1
	`C_generateKey`	540×86^8	512	69	8	1
	`C_setAttributeValue`	86	14	3	1	1
	`C_encrypt`	20	3	4	2	1
Atos CardOS V4.3	`C_logIn` 1	1	1	1	1	1
	`C_sign`	1	1	1	1	1
	`C_findObjects`	1	1	1	1	1
	`C_getAttribute`	1	1	1	1	1
	`C_setAttribuyeValue`	1	1	1	1	1
Atos CardOS V4.4	`C_logIn` 1	1	1	1	1	1
	`C_sign`	1	1	1	1	1
	`C_findObjects`	1	1	1	1	1
	`C_getAttribute`	1	1	1	1	1
	`C_setAttribuyeValue`	1	1	1	1	1
Atos CardOS V5	`C_logIn` 1	1	1	1	1	1
	`C_sign`	1	1	1	1	1
	`C_findObjects`	1	1	1	1	1
	`C_getAttribute`	1	1	1	1	1
	`C_setAttribuyeValue`	1	1	1	1	1
RSA SecurID	`C_logIn`	7396	65	39	21	1
	`C_findObjects`	7396	6	1	1	1
	`C_getAttributeValue`	54700816	3	1	1	1
	`C_sign`	86	1	1	1	1
Safesite TCP ISV1	`C_logIn`	1	1	1	1	1
	`C_sign`	12322	53	7	4	2
	`C_setAttributeValue`	1	1	1	1	1

Table 5.4: Reduction in the number of alternative implementations during the analysis.

continuously restricted until the final model is produced. To demonstrate REPROVE's effectiveness on narrowing-down, we have implemented a baseline algorithm that generates a search tree that consists of all possible mappings (including different interpretations of each command) of the APDU trace, based on the category each command belongs to. Table 5.4 presents the command combinations produced by the baseline algorithm, termed *B.CC*. The terms *R.CC*, *R.SBC* and *R.FC* present REPROVE's total command, sub-functionality and functionality combinations respectively. *Model* is the number of final model(s) suggested by REPROVE for the specific cryptographic function. At each successive step the number of alternative implementations is progressively reduced. As Table 5.4 demonstrates there are cases that *B.CC* is 1. That happens when either most of the commands or all the commands of the trace are inter-industry, which suggests an 1-1 mapping. However, in some cases the search space is prohibitive, eg., in Athena ASEKey USB for the `C_generateKey` function there are 540×86^8 total command combinations. Even in such cases REPROVE narrows-down the combinations to a single mapping.

Discussion.

REPROVE inferred at least a high-level model of the actual implementation for all tested cards. In some cases the reverse-engineering outcome was more than one model, each one capturing the same on-card operations but differed at the implementation level. For example read data from binary files versus read data from records. We do not consider this as a failure as REPROVE provided at least a high-level view of the implementation.

However, this shows the variety of the implementation tactics in the different vendors. For example, there does not exist a uniform way for handling key related data or the particular authentication mechanisms to be used. Each analysed card had its own unique implementation. The APDU layer offers a general purpose mechanism for calling on-card operations which can be refined to the particular assumptions and needs of each vendor. Being able to guarantee 1-1 mappings addresses the necessity of incorporating feedback techniques to refine the reverse-engineering outcome. Although, if the outcome suggests the same abstraction of different implementations, manual inspection to identify the correct one is a trivial process.

5.5 Summary

This chapter presented REPROVE, a system for automatically analysing the low-level communication protocol of a smart-card by reasoning over a formal model of the ISO 7816 standard, regardless of the protocol's implementation. We have used REPROVE to successfully extract at least one model from each tested card and have shown that, although analysing proprietary implementations is a combinatorial problem, it is possible to leverage background knowledge to effectively reduce the search space. To the best of our knowledge, REPROVE is the first system that successfully reverse-engineers proprietary APDU implementations. REPROVE's results can provide the necessary evidence to reason about the implementation of the protocol and discover possible security flaws. Obtaining such evidence is especially crucial, as bad implementations may lead to fraud and/or disputes between card issuer and client. Detecting such violations manually is not trivial: it requires either knowledge of the semantics of the communication trace, access to the PKCS#11 library or/and to the card's code. REPROVE does not have such prerequisites.

Reverse-engineering PKCS#11 based APIs and discovering vulnerabilities is not a new idea, *e.g.*, Tookan [Bortolozzo et al., 2010] reverse-engineers a card's API and discovers security flaws with respect to the standard. On another perspective, Caml Crush [Benadjila et al., 2014] acts as an attack filtering tool that sits between the PKCS#11 device and the calling application. Caml Crush considers attacks only at the API level and not at the low-level communication. Targeting the implementation of PKCS#11 at the low-level communication is a novel idea and suggests a new way of attacking the standard by bypassing the API and talking directly to the device, thereby avoiding API-level restrictions. Such attacks cannot be detected nor filtered by such tools, as they address strictly the API level. REPROVE addresses PKCS#11 attacks at the APDU layer. PKCS#11 defines specifications for secure implementations and applies to a broad range of cards. These specifications have to be addressed at the communication layer as well, *e.g.*, in session identification.

As we present in the following chapter (Chapter 6) REPROVE's analysis allowed us to expose severe violations of the standard's specifications. Reaching these findings in the first place would not have been possible without reverse-engineering. We therefore believe our approach cuts across all layers of the PKCS#11 implementation and provides a blueprint that can be applied to other models and protocols as well.

Chapter 6

Security Analysis on the Reversed-Engineered Smart-cards

In the smart-card setting, establishing security and compliance with a standard is a difficult task. A common perception is that by incorporating a well-defined and verified standard the whole smart-card setting becomes secure. Proprietary implementations that only focus on providing the required functionality in combination with a lack of means to verify them commonly leads to unwanted behaviour, logical flaws and, most importantly, to security vulnerabilities; it is common that an implementation that provides more functionality than the desired one may be open attack to vectors that exploit that extra feature.

A potential approach to verifying an implementation would be formal verification, a widely used process of checking whether the desired specifications are met and proving correctness of the underlying algorithms with respect to the implemented standard. This approach requires the existence of well-defined formal models that capture the intended behaviour. By definition, a model is formal if it has unambiguous mathematically defined syntax and semantics. However, the lack of universally accepted translation of the PKCS#11 API code to the smart-card communication, results in proprietary models.

6.1 PKCS#11: Threat Modelling for PKCS#11 Security Tokens

The most well-known modelling technique for verifying cryptographic protocols is the Dolev-Yao model [Dolev and Yao, 1983] which proposed a way of modelling the protocols and the attacker to allow automated verification of the protocol's security. The original intention of the model was to target public-key protocol analysis against active attackers that have complete control over the network and can i) act as a legitimate user and obtain any message from the network, and/or ii) can initiate the protocol with any party on the network. Dolev-Yao proposed the algebraic abstraction of the protocol by modelling bit strings as terms, cryptographic operations as functions over those terms and the attacker as deduction rules on the abstract algebra. Although deducing many details from the protocol leads to un-discoverable or false attacks, the Dolev-Yao abstract modelling enhances the verification process.

The main goal of PKCS#11, as stated in the standard [RSA Security INC, 2004], is to provide a secure API for hardware devices in which private objects are protected. PKCS#11 provides a set of functions for managing private objects, which are characterised by their *attributes*. The role of an object is described by its attributes, and any operation on that object must comply with that role. For example, the *sensitive* and *unextractable* attributes denote a protected object. Clulow [Clulow, 2003] was the first to address security issues in PKCS#11 by introducing a set of attacks according to the following generic API threats: i) a malicious security officer who abuses the authority of his position and his access to the device and consequently to the user management functions; ii) a malicious user who exploits his authorised access to the token; iii) a malicious third party who gains access to the token through some other means. These threats describe the capability of an attacker to initiate a PKCS#11 session, relay a session with the device or have access to the user's credentials (PIN).

Literature in the security analysis of APIs and particularly of PKCS#11, *e.g.*, [Delaune et al., 2008, Bortolozzo et al., 2010, Delaune et al., 2010], uses the Dolev-Yao modelling process to formally reason about security threats similar to the ones introduced by Clulow. The attackers' abilities are extended, from just modifying messages to also initiating API calls with those messages. In the PKCS#11 setting the attacker is able to perform any cryptographic operation once he gains access to the corresponding key [Centenaro et al., 2012]. The attacker can obtain access to that key through testing different parameters via API calls. Under the Dolev-Yao analysis, the attacker can arbitrarily decompose and recompose messages, under the assumption that he can only decrypt encrypted messages if he possess the key.

The security requirements of a smart-card based system and the potential threats have long been an open-ended discussion, where the possible vulnerabilities vary depending the utilised attacker model. [Schneier et al., 1999] is one of the first efforts to address threat scenarios for smart-card based systems. The authors identify the attacker's goals based on his role within a system: the attacker is the terminal and targets the card-holder, the attacker is the card-holder and targets the terminal, the attacker is the card-holder and targets the data owner, the attacker is the card-holder and targets the issuer, the attacker is the card-holder and targets the software manufacturer, the attacker is the terminal and targets the issuer, the attacker is the issuer and targets the card-holder and finally, the manufacturer is the attacker and targets the data owner. The attacker may be legitimately given a specific role or he may obtain it with illegitimate methods.

In [De Cock et al., 2005] the authors address the security issues that may arise in smart-cards for web applications such as electronic identities. They categorise them depending on the targeted participating party, as each party may provide an attack entry point. They consider attacks that target the behaviour of the smart-card reader, malware threats that take advantage of the smart-card driver to gain access to the system, attacks against the Crypto API or the smart-card API to compromise authentication secrets, attacks that exploit the user's privileges on the system and Denial of Service (DoS) and spoofing attacks against the web servers.

The security of PKCS#11 smart-cards has been only addressed in [Bozzato et al., 2016], which defines a threat model that addresses the APDU layer. The model defines the following optional attacker capabilities:

- full access to the application;
- full access to the application except from the authentication mechanism which is controlled by a separate software/hardware;
- no access at all due to low-level restrictions.

The attack scenarios this model considers are: i) accessing the user PIN, ii) performing cryptographic operations, iii) gaining access to sensitive keys. The adversary's goals and capabilities follow the same line as the ones defined in the PKCS#11 API threat models; the difference is that the adversary controls the APDU communication instead of the network communication.

Abstracting the APDU Threat Model Identifying possible threats at the low-level communication can be perceived as a more complex procedure compared to threats at the API or the network layer. As discussed in Chapter 5.2, we consider each APDU session to be defined by three abstractions:

1. a PKCS#11 function;
2. a set of smart-card operations;
3. a communication session (APDU command).

Based on this abstraction, different vulnerabilities are addressed. For example, a threat model that considers only PKCS#11 will examine violations of the standard, faulty key management and exploitable execution of the cryptographic functions. A threat model that considers the smart-card operations, independently of the higher-level protocol that is implemented (*i.e.*, PKCS#11), will address issues such as lack of authentication mechanisms and exposure of protected data and operations. Finally, a threat model for communication will not consider violations of the higher-level protocol (*i.e.*, PKCS#11), nor possible exploits on smart-card operations; it will address issues such as plaintext communication, leakage of secret and private data, and replay attacks. The threats that are discussed in the following section are goal-driven with respect to each abstraction of the APDU layer.

A threat model aims at identifying possible implementation exploits. Because of the proprietary nature of the PKCS#11 APDU implementations, to address the attacks that each smart-card is vulnerable to, requires an individual threat-model for each smart-card. The goal of this section is to address possible attack scenarios and patterns, rather than mitigations of the attacks. By doing so, we are able to identify a set of vulnerable APDU parts that apply to every implementation. We introduce a goal-oriented threat model that identifies the potential attack scenarios on PKCS#11 smart-cards. Our threat model is based on the attacker's goals; depending on the goal a different range of attack scenarios is identified. Finally, we narrow-down the vulnerable parts to a specific set that can be used to mitigate the identified attack-scenarios.

6.1.1 Threat-Modelling Techniques

Threat-modelling is the process by which potential threats within a system are identified and assessed. The most commonly used techniques are the STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of interest), the PASTA (Process of Attack Simulation and Threat Analysis), the TRIKE and the VAST (Visual, Agile, and Simple Threat modelling) methodologies.

The STRIDE technique provides a threat classification scheme against the security properties of a system. The goal is to assess Confidentiality, Integrity and Availability (CIA) along with Authorisation, Authentication and Non-Repudiation. STRIDE [Swiderski and Snyder, 2004] modelling follows an attacker goal-driven approach for identifying the possible threats. PASTA [UcedaVelez and Morana, 2015] aims at providing a modelling technique which aligns business objectives with technical requirements. PASTA incorporates business impact analysis with compliance requirements in an attacker-centric approach to identify potential threats. The TRIKE [Saitta et al., 2005] technique identifies threats following a risk-management approach. The process begins with a *requirements model*, which identifies the parameters that the system should satisfy, and assigns different levels of risk to each asset of the system. The VAST technique classifies the threat models into application and operational/infrastructure. The application threat models (ATM) address vulnerabilities with regards to the different features and use cases of an application. The operational threat models (OTM) assess the security of the different independent, grouped and shared components of a system.

To define a threat model for the APDU layer of PKCS#11 smart-cards, PASTA is the least appropriate technique of the aforementioned as it is business-driven. TRIKE modelling, being a risk-based approach, requires a well-defined model of the requirements that the APDU layer should satisfy. However, the security conditions that the APDU layer should satisfy are obscure. The VAST technique is used to analyse the potential threats in larger settings, whereas a system can be decomposed and analysed individually. To assess the possible vulnerabilities of the APDU layer we chose the STRIDE threat classification. STRIDE being a goal-driven approach allows us to identify different attack scenarios with respect to the attacker and the general properties that the smart-card setting should have: the information flow of the setting should remain secret to an unauthorised entity; the exchanged messages have not been tampered and come from the original source; the setting is always available and behaves as expected by a legitimate user.

STRIDE is a popular goal-driven technique used to discover possible vulnerabilities and weaknesses of a system. The STRIDE model is widely used for the security analysis of software systems and to generate threat scenarios for automated testing. For example in [Xu and Nygard, 2006] the authors employ the STRIDE categories to identify threats which they formalise and use for software verification. In [Xu et al., 2012] the STRIDE modelling process is used for identifying possible vulnerabilities of the Magento (a web-based shopping system) and FileZilla (an FTP server implementation); based on the identified vulnerabilities automated security tests are generated.

6.1.2 Attack Scenarios

The STRIDE model defines the following six categories of attacks:

1. *Spoofing*: the attacker impersonates a legitimate entity of the system.
2. *Tampering with data*: the attacker tampers with the data in an unauthorised way, regardless of whether the data is persistent or changes at each session.
3. *Repudiation*: a legitimate user denies performing an action that the system expected by him, but there is no way to prove this denial.
4. *Information disclosure*: a legitimate user or an attacker gains unauthorised access to data.
5. *Denial of Service*: deny a service to the remaining parties within a system.
6. *Elevation of privilege*: the attacker gains privileged access to a system which otherwise needs special authorisation.

Our adversary model assumes perfect cryptography: we do not consider vulnerabilities on the implemented cryptographic algorithms, *e.g.*, low-entropy key generation and timing attacks on private key operations. Such vulnerabilities are out of the scope of our work; our goal is to identify threats at the communication layer. The setting we consider consists of the communication between the API and the smart-card, under a single-user configuration. The communicating parties request operations from each other and exchange data. We categorise the adversary's goals to i) short-term: the adversary targets a single session or requires access to the smart-card only once., ii) long-term: the adversary targets more than one session, or requires repeated access to the token. Finally, as the attacks target only the communication between the API and the smart-card, physical and API attacks are not discussed.

Spoofing The attacker can impersonate i) the API to the token, and/or ii) the token to the API. The `C_logIn` function is responsible for validating an entity, based on a shared secret. Depending on the implemented protocol, either only the API or both the API and the token have to authenticate using the shared secret: i) the API authenticates itself to the token by proving the knowledge of a PIN; ii) the API and the token mutually authenticate by proving the knowledge of a shared key. For an attacker to be able to spoof either the API's or the token's identity, he needs access to the shared secret. The following scenarios identify possible ways to achieve that.

S.1 The API-token authentication is PIN based and the PIN is sent in plaintext. The token verifies the correctness of the PIN and notifies the API that the session may continue as usual. i) If the adversary's goals are short-term, the attacker is not interested in possessing the PIN but in accessing a specific session. The adversary has now got access to the desired session through the user's authentication and may proceed to pursuing his goal. ii) If the adversary's goals are long-term, he can replay the authentication protocol and initiate a session with the token.

S.2 The API-token are mutually authenticated based on a session key. i) If the attacker's goals are short-term, the attacker may proceed to pursue his goals after the authentication protocol. ii) If the attacker' goals are long-term he can only access a session if the API has initiated.

Tampering with Data An attacker may alter, insert, delete or re-order the exchanged data. The attack scenarios that we identify are independent on the duration of the adversary's goals; the same strategy applies to both short-term and long-term goals. For the smart-card setting we consider two categories of data that might be vulnerable to tampering: static and dynamic.

1. Static data: the data that is stored in the smart card: the keys, the key attributes and handles, the cryptographic algorithms and the certificates.

2. Dynamic data: the data that is transmitted during the communication. Dynamic data refers to: i) the command, *i.e.*, the message that is sent by the API; ii) the response, *i.e.*, the message that is sent by the token; iii) the message data, *i.e.*, the data part of a command or a response.

T.1 The attacker tampers with the data parts of the commands with respect to the executed operations:

i) during the `C_generateKeyPair` function which requests the generation of the public/private key pair $\{sk_s, pk_s\}$, (the same applies for the `C_generateKey` function) the adversary substitutes the attributes $\{att_1, att_2, ..att_n\}$ of the key sk_s or sk_a respectively to $\{att_{a1}, att_{a2}, ..att_{an}\}$;

ii) during the `C_encrypt` function which requests the encryption a message m with a symmetric key sk, the adversary substitutes m with m_a;

iii) during the `C_sign` function which requests the signature of a message m with the private key sk, the attacker substitutes m with m_a;

iv) during the `C_setAttributeValue` function which requests the modification of the attributes of a key, $att_1 \mapsto att_2$ the attacker alters att_2 to att_a so that the token performs $att_1 \mapsto att_a$.

v) during the `C_unwrapKey` function which requests the unwrapping of a wrapped key $\{sk\}_k$, the attacker substitutes the wrapped key $\{sk\}_k$ with $\{sk_a\}_k$.

T.2 The attacker tampers with the token's responses with respect to the executed functions:

i) during the `C_encrypt` function which requests the encryption of a message m with the symmetric key sk, the adversary returns the computation $\{m\}_{sk_a}$ with his own sk_a;

ii) during the `C_decrypt` function which requests the decryption of $\{m\}_{sk}$, the adversary substitutes the response m with m_a.

iii) during the `C_sign` function which requests the signature the signature of the message m, σ_{sk}, with the private key sk, the adversary substitutes the response $\sigma_{sk} \mapsto \sigma_{ka}$ where σ_{ka} is the signature of the the message m with his own key ka.

T.3 The attacker initiates his own session with the token and tampers with the token's data. For example, by i) altering the attributes $\{att_1, att_2, .., att_n\}$ of the key; ii) altering the handle of a key; iii) deleting a key; iv) changing the PIN.

Repudiation Except for the key-exchange protocol, the token is the communicating party that performs operations. If the operation is not supported, or the provided data is incorrect, then the token will respond with an `error` code. Repudiation can occur when the attacker returns the expected response to the API, *e.g.*, the outcome of a cryptographic function, even if the token has responded with an error.

R.3 Whenever the API requests a key the attacker returns his own key. Also, whenever the API requests the computation of a cryptographic function, the attacker returns an output computed with his own keys. For example, for the symmetric encryption scheme the attacker has provided his own sk_a to the API. Whenever the API requests the encryption of a message m with the symmetric key sk, the attacker responds with $\{m\}_{sk_a}$. Such attacks can occur on a single session (short-term adversary goals), or in multiple session (long-term adversary goals).

Information Disclosure Information disclosure, or privacy loss, describes situations in which an adversary gains access to *private* data. Table 6.1 presents the data that we have identified as *private* within the smart-card setting. If data that falls to this category becomes exposed, it opens an attack vector to the corresponding directly connected entity. For example, i) exposure of the PIN entails privacy loss for the user; ii) exposure of the public keys might lead to linkability of encrypted and/or digitally signed messages; iii) exposure of the private/secret keys poses a threat to the token's and consequently to the user's privacy; iv) the attributes and the handle of a key are private information about that key and their exposure may open an attack vector for the key itself; v) the plaintext messages (before their encryption, after their decryption and before being digitally signed) are private information of the user; vi) digital signatures may be linked to a single user.

I.1 Information disclosure issues arise whenever an attacker has access to one or more of the data that appears in Table 6.1, in a single (short-term adversary goals) or multiple sessions (long-term adversary goals).

Data	**Entity**
PIN	user
public keys	user
private/secret keys	user, token
key attributes	user, token
key handles	user, token
message to be encrypted	user
decrypted message	user
message to be signed	user
signed message	user

Table 6.1: Categorisation of private data with respect to the corresponding entity.

Denial of Service An attacker may alter parts of the data that is exchanged during a session, or parts of the data that is already stored in the token, in a way that causes the token's failure to execute specific operations or to produce an erroneous for the user outcome. The data that, if tampered, may cause a Denial of Service (DoS) attack is the one presented in Table 6.1. For example, given a private key sk_1 with the corresponding handle h_1, a change to the key handle such that $h_1 \mapsto h_2$ whereas h_2 is an invalid handle, causes the token's incapability of using sk_1. DoS attacks are caused by tampering with the data in a faulty way and may occur in a single or multiple sessions. This can be achieved by:

i) tampering with the message flow: the attacker alters or deletes command/responses;

ii) tampering with the exchanged data: the attacker alters or deletes parts of the exchanged data;

iii) tampering with the data stored in the token: the attacker alters or deletes parts of data that is stored in the token and is essential for performing operations.

DoS attacks target:

i) the user's ability to use the token: either by making it impossible for the user to login, or making it impossible for the user to execute cryptographic functions;

ii) the applications that require as input the token's computations.

We categorise the possible ways that an attacker can mount a DoS attack based on the signature scenario presented in Example 6.1.1. The same scenarios apply to simple cases *e.g.*, the user authentication to the token, to more complex ones *e.g.*, the generation of a key pair.

Example 6.1.1. *The user has requested the signature of m with the private key sk. The key's handle is handle h, and is assigned to the `CKA_SENSITIVE` and `CKA_SIGN` attributes*[1]*. The verification key is pk. The signature is requested by the following sequence of commands:*
{`select(h)` $\rightarrow$ `OK`, `sign(m)`$\rightarrow$ σ_{sk} },
where select(h) provides the handle of sk, f `sign(m)` requests the signature of m in which the token responds with the corresponding signature σ_{sk}. The order of the commands is essential for the token to be able to interpret the signature request correctly.

D.1 The adversary relays this communication by withholding all or some commands to be sent to the token. For instance, in Example 6.1.1, the attacker will cause

[1]The keys that are used to sign messages must always have this attributes set to TRUE.

a DoS attack by not letting $select(h)$ to be sent to the token. Since no private key has been defined for the signature, the token will abort the process and will respond with an `error` code.

D.2 The adversary relays a session and tampers with the data that is exchanged, in a way that the token cannot interpret. For instance, in Example 6.1.1 the attacker performs the following substitution in the exchanged commands: $h \mapsto h'$ where h' does not correspond to a valid key. Since no private key has been defined for the signature, the token will abort the process and will respond with an `error code`.

D.3 The adversary relays a session and tampers with the data that is exchanged in a way that will cause the token to produce a faulty outcome. For instance, in Example 6.1.1 the adversary relays the communication so that σ_{sk} is substituted by σ_{ka} (a signature with the adversary's key k_a). The attack takes effect when the verification of σ_{ka} with pk will fail.

D.4 The adversary has access to the token and changes the parts of the stored data in a way that makes them inaccessible by specific operations. In Example 6.1.1 an adversary can mount a DoS attack by setting `CKA_SIGN` to `FALSE`, as sk can no longer be used for signatures.

Elevation of Privilege PKCS#11 defines two categories of users: the normal user and the Security Officer (SO). The SO user can access only public objects; however, he can perform privileged functions such as setting the user's PIN. The SO user is logged-in only during the initialisation of the token. Although not all tokens support SO users, in case they do the SO user logs-into the token by using a PIN. The attack scenario in which an attacker gains access to higher privileges than a normal user is described in *E.1* and only applies if the adversary has access to a particular session (short-term adversary goals).

E.1 The adversary controls the communication channel during the token's initialisation. The API sends the SO PIN in plaintext, which the attacker sniffs.

6.1.2.1 Identifying Attack Patterns

The threat model that we defined identifies the possible attack scenarios based on the attacker's capabilities and goals. Each attack may target different parts of the communication *i.e.*, PKCS#11 functions, token operations, particular parts of the communication protocol, and/or different categories of data *e.g.*, exchanged during a session or already stored in the token. Since the PKCS#11 low-level implementations do not comply with a standard, the mitigation of an attack is dependent on the underlying implementation of each smart-card. For example, consider an attack in which the adversary extracts the private key *sk*. Given two tokens from different vendors, $token_1$ with sk_1 and $token_2$ with sk_2, the adversary extracts sk_1 via the *wrap-decrypt* attack and sk_2 via a *select-read* attack. This example demonstrates that the adversary's goal may be reached through different steps. The attack vectors and their mitigation being so broad makes it difficult to reason about the security of tokens from different vendors. Thus, identifying commonalities between the attacks is necessary.

We specify the communication parts that are exploitable to the aforementioned attacks. We show that this knowledge is reduced to specific parts of the communication and that vulnerability testing can be achieved without the need of creating individual attack scenarios for each token. Table 6.2 summarises the attack categories introduced in Section 6.1 and the communication parts that such attacks target.

	PIN	*sk*	att_i	*h*	$\mathbf{m}\rightarrow \{m\}_{sk}$	$\{m\}_{sk}$	$\{m\}_{sk} \rightarrow\mathbf{m}$	$\mathbf{m}\rightarrow \sigma$	σ	*f*
S.1	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
S.2	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
T.1	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗
T.2	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗
T.3	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
R.3	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓
I.1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
D.1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
D.2	✓	✗	✓	✓	✗	✗	✓	✗	✗	✗
D.3	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗
D.4	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗
E.1	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table 6.2: The identified attacks and the data parts they exploit.

As Table 6.2 illustrates, the vulnerable parts of the communication that we identified are the following: i) *PIN*: allows access to the token's functionalities, ii) sensitive key (private/secret) *sk*: allows the computation of the PKCS#11 functions outcomes,

iii) key attribute(s) att_i: allows the modification of a key, iv) key handle h: allows the modification of the way a key is accessed, v) message to be encrypted $\mathbf{m} \to \{m\}_{sk}$: allows access to message m, vi) encrypted message $\{m\}_{sk}$: allows access to an encrypted message, vii) decrypted message $\{m\}_{sk} \to \mathbf{m}$: allows access to the decryption result m, viii) message to be signed $\mathbf{m} \to \sigma$: allows access to message m, ix) signature σ: allows access to σ, x) PKCS#11 functions f: allow access to PKCS#11 functions and to the token's operations. The set corresponds to the data that is compromised by the discussed attack scenarios. In the next section we discuss the security of the smart-cards that we have previously reverse-engineered using REPROVE, with respect to these communication parts.

6.2 Manual Analysis of the Reversed-Engineered Models

We consider the threat model presented in Section 6.1 and in particular whether the APDU implementation of each smart-card allows the data presented in Table 6.2 to be exploited. We analyse each implementation, as modelled by REPROVE, based on the three abstractions of the APDU layer: i) PKCS#11, ii) smart-card operations, and iii) APDU communication. The tested PKCS#11 functions, as presented in Table 5.3, are a mixture of sensitive and not sensitive operations that deal with private and secret keys.

Atos CardOS : All three smart-cards share exactly the same implementation. The authentication mechanism is PIN based and the PIN is sent in plaintext. The communication proceeds in plaintext as it is not protected by any security mechanism, such as secure messaging. The inferred models did not suggest faulty management of the keys such as leakage of the key's value.

Aladdin eToken Pro : The implemented authentication mechanism follows a challenge-response protocol based on a session key. The communication proceeds in plaintext as it is not protected by any security mechanism, such as secure messaging. The inferred models suggested a faulty key management on secret keys for the `C_generateKey` and the `C_wrapKey` functions. The implementation of the `C_generateKey` function suggests that the key is generated API side and then stored in the card: the inferred executed operation is a single `store_data`. The implementation of the `C_wrapKey` function suggests the value of the key to be wrapped is returned: the inferred executed operation is a single `read_data` operation. Since secure messaging is not implemented, the values of the keys are sent in plaintext.

Athena ASEKey USB : The implemented authentication mechanism follows a challenge-response protocol based on a session key. The communication is protected by the secure messaging protocol only when the `C_generateKey` and `C_generateKeyPair` functions are executed. For the remaining functions the communication proceeds in plaintext. The inferred models did not suggest faulty key management such as leakage of the key's values.[2]

RSA SecurID USB : The implemented authentication mechanism follows a challenge-response protocol based on a session key. The communication proceeds in plaintext as it is not protected by any security mechanism, such as secure messaging. The inferred models did not suggest a faulty key management.

Safesite TCP ISV1 : The implemented authentication mechanism is PIN based, whereas the PIN is sent in plaintext. The communication proceeds in plaintext as it is not protected by any security mechanism, such as secure messaging. The inferred models did not suggest a faulty key management.

We summarise the vulnerabilities suggested by the inferred models, with respect to the tested PKCS#11 functions and the data that can be exploited.

(*i*) *PIN*: The authentication protocol follows a PIN-based verification, whereas the PIN is sent in plaintext. The implementation allows an attacker to mount the *S.1*, *S.2*, *S.3*, *T.1*, *T.3*, *I.1*, *D.2*, *D.4*, *E.1* attacks. The smart-cards that fall in this category are the Atos CardOS V4.3, Atos CardOS V4.4, Atos CardOS V5 and Safesite TCP ISV1.

(*ii*) *sk*: The implementation allows access to a private/secret key and is consequently vulnerable to the *S.3*, *T.1*, *T.3*, *I.1* attacks. Aladdin eToken Pro exposes the value of a secret key at the `C_generateKey` function and at the `C_wrapKey` function.

(*iii*) att_i: The location and the values of the attributes of the tested private/secret keys were exposed. Consequently, the implementation allows an attacker to mount the *S.3*, *T.1*, *T.3*, *I.1*, *D.2*, *D.4* attacks. In all smart-cards the file that stores a key's attributes are exposed. The attribute values of a key are also revealed.

(*iv*) *h*: The handle and where it is stored, of each tested key, were exposed. The

[2]In [Bozzato et al., 2016] the authors manually reverse-engineered the secure messaging protocol and discovered the used session key. After decrypting the data parts of the communication, they concluded that the secret key is generated API side and sent to the token. The analysis we present is purely based on REPROVE's inferred models. Although reverse-engineering the encrypted parts of the communication provides a better understanding of the implementation, brute-forcing the different implemented mechanisms it is not within the scope of our work.

implementation is prone to the *S.3*,*T.1*, *T.3*, *I.1*, *D.2*, *D.3*, *D.4* attacks. In all smart-cards the handle and the file that is stored is exposed.

(*v*) $m \rightarrow \{m\}_{sk}$: The message to be encrypted is not protected by any security mechanism. The implementation is prone to the *T.1*, *T.2*, *I.1*, *D.3* attacks. All tested smart-cards are vulnerable to those attacks.

(*vi*) $m \rightarrow \sigma$: The message to be signed is not protected by any security mechanism. The implementation is prone to the *T.1*, *T.2*, *I.1*, *D.3* attacks. All tested smart-cards fall in this category.

(*vii*) σ: The signature is transmitted without being protected by a security mechanism. The implementation is exposed to the *S.3*, *I.1* attacks. All tested smart-cards fall in this category.

(*viii*) f: The sensitive operations are only protected by the user's PIN. The implementation allows an attacker to mount the *S.3*, *I.1*, *D.1* attacks. Atos CardOS V4.3, Atos CardOS V4.4, Atos CardOS V5 and Safesite TCP ISV1 fall in this category.

6.3 Automated Extraction of State-Machines

We present a semi-formal systematic analysis of the tokens with regards to their inferred implementation, in which we extract the smart-card's state machines via hardware interaction. The goal is to obtain models that can indicate whether the analysed implementation introduces exploitable security weaknesses.

Extracting meaningful models requires access to the smart-card as well as familiarity with the communication semantics. Additionally, the analysis of the extracted models requires knowledge of the on-card operations and how they can be initiated. REPROVE is designed to provide such insight. We integrated REPROVE with the SmartCardLearner system [Ruiter, 2015], which uses *regular inference*, also known as automata learning, to extract state-machines of different implementations. We created an ecosystem system that translates and analyses the PKCS#11 implementations at the APDU layer.

Figure 6.1 presents how the two systems interact. Given a sniffed PKCS#11 function, REPROVE infers the semantics of the low-level communication, and the on-card operations. When the reverse-engineering process finishes, REPROVE provides a translated trace to SmartCardLearner and initiates the analysis. The result is a state machine of the sniffed session with regards to REPROVE's output.

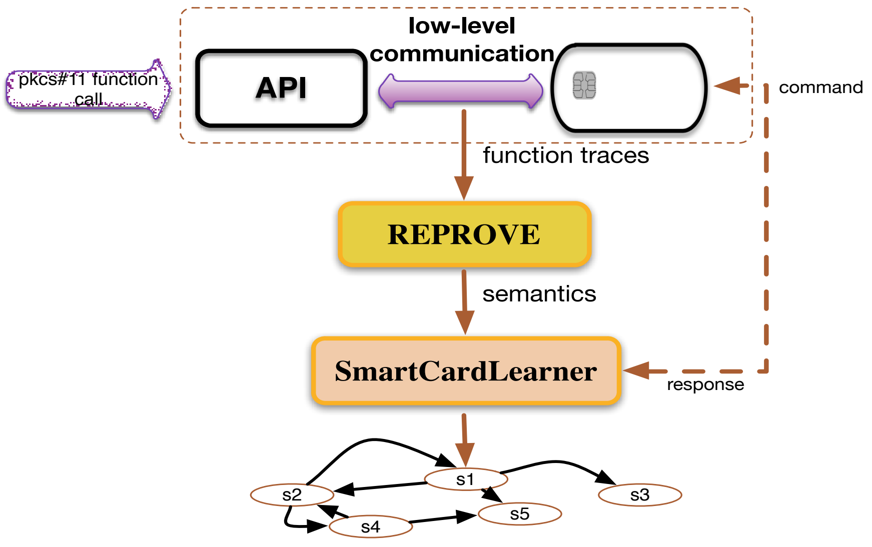


Figure 6.1: Reprove and SmartCardLearner interaction.

SmartCardLearner In [Fides Aarts, 2013] the authors presented a technique for using machine learning in order to extract state machines of EMV smart-cards. SmartCardLearner, given a set of valid commands and their appropriate instantiations, sends all possible combinations of the commands to the card to learn the different states caused by the commands. The system is based on LearnLib [Raffelt et al., 2005] and needs to be provided with a list of commands that will be tested against the smart-card. As such, the exact set of the commands that the smart-card can interpret is needed. The system concludes a state-machine by testing different sequences of commands and examining the corresponding responses.

SmartCardLearner Input and Output To use SmartCardLearner an input alphabet of the commands used by the card needs to be provided. The input is automatically constructed by REPROVE, which also initiates the analysis process. An example input used while testing the Atos Cardos V4.4 smart-card is presented in Figure 6.2. Each command is described by a label, for example the label `PERFORMSECOP` identifies the `perform security operation` command. Each command is provided with its exact instantiations, *i.e.*, the `P1`, `P2`, `Lc`, `D`, and `Le` fields. The label of each command must be unique with regards to specific instantiations, in order to get a more detailed output; if the same command appears with different instantiations, then a different label is used. For example, as presented in Figure6.2, although `SELECTa` and `SELECTb` refer to the same command (`CLA=00`, `InS=00A4`), separate labels are used to distinguish the different instantiations.

```
SELECTa;00A4080C0450154400
READa;00B20200EA
SELECTb;00A4080C08501550724B025502
MANAGESECENVA;00220301
MANAGESECENVB;002201B803840102
PERFORMSECOP;002a808681000001f...80
```

Figure 6.2: SmartCardLearner sample input for the Atos CardOS V4.4 smart-card.

The commands that are used for the testing have exactly the same form as they appear on the sniffed traces. One could argue that the system should test different instantiations of the commands. However, to identify the instantiations that are accepted by the card, fuzzing with all possible instantiations and reverse-engineering of the structure and the meaning of the exchanged data is required. However, this process requires manual inspection and interaction with the API as the semantics of each command would still remain unknown; the only feedback of each instantiation would either be a positive (9000) or an ERROR response. Such process is time consuming and requires complex technical skills. An example of such process is discussed in Section 6.4.

SmartCardLearner tests the input commands by exhaustively trying all possible permutations and outputs the corresponding state-machine. For example, given the input presented in Table 6.2, SmardCardLearner outputs the state-machine presented in Figure 6.3. The output consists of all different states that can be reached given the input commands. A command may alter the state only if the following preconditions are satisfied: i) the response should not contain an error code, *i.e.*, should be 9000, and ii) the command is a prerequisite to that state, *i.e.*, it cannot be skipped.

Targeted Functions to Test The purpose of this work is to test whether the data that appears in Table 6.2 can be exploited, *i.e.*, whether an attacker can gain the necessary knowledge to mount one or more of the attacks discussed in Section 6.1. The complexity of the extracted state-machines depends on the SmartCardLearner's input. Testing all analysed PKCS#11 functions at a single run results in huge diagrams which are difficult to manually process (an example of such a diagram is presented in Figure B.1 in Appendix B), making it almost impossible to identify flaws. However, as previously discussed, to conclude on possible security vulnerabilities we only need to test a subset of functions which provide an insight on i) the *authentication secret*, ii) the

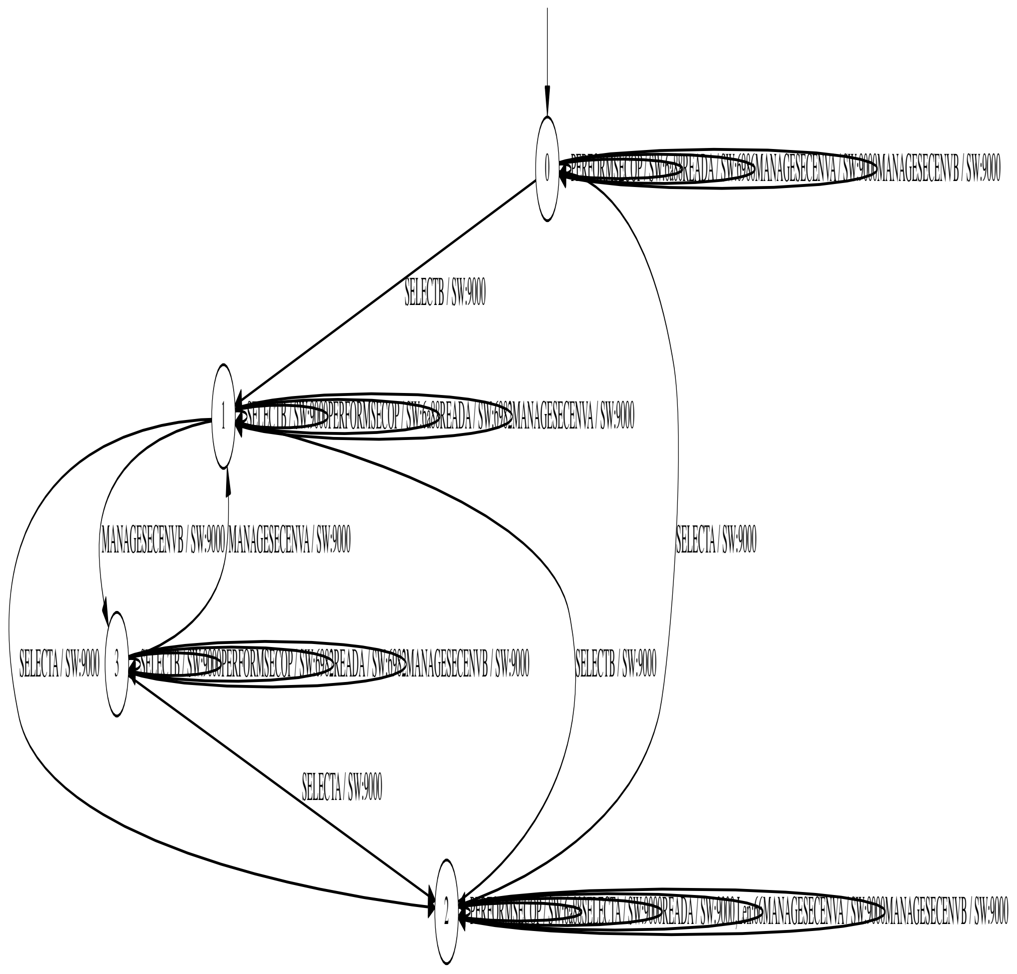


Figure 6.3: The Cardos 4.4 state-machine given the input presented in Table 6.2.

management of a private/secret key *sk*, iii) the protection mechanisms of the key attributes $att_1, att_2, .., att_n$ and handle *h*, iv) the protection mechanisms of the messages *m* to be signed/decrypted/encrypted and the corresponding result, and v) the protection mechanisms of the PKCS#11 functions f. Given a sniffed trace with known semantics, the tests that are performed are the following:

- Session Replay test: replay a PKCS#11 session to check whether session handles or other countermeasures are used that prevent replaying even if the user's credentials have been exposed. If the test succeeds, it proves that an attacker can have access to the *authentication secret* and to the token's functions f. Also, a successful replay of a session that performs a cryptographic operation over some message m, implies that m is sent in the clear; if m was encrypted with a session key then replay would not be possible. Consequently, this test also proves that the attacker can have access to m and the result of the cryptographic function over m.

- Lack of Authentication test: replay a PKCS#11 session without including the authentication part. A successful test proves proves that the PKCS#11 functions are not protected. Therefore, the attacker can have access to the token's functions f (and consequently to m), and he is also considered to posses the necessary *authentication secret*[3].

- Alteration of Key Attributes test: given a key sk (private/secret) with handle h and attributes $att_1, att_2, ..., att_n$, initiate the corresponding PKCS#11 session which alters $att_m \in att_1, att_2, ..., att_n$. If the test succeeds, it proves that the key attributes and handle[4] are not protected and can be randomly changed. Access to $att_m \in att_1, att_2, ..., att_n$ and h may ultimately provide access to the corresponding key, sk, through key-separation and redirection attacks.

We only tested the `C_logIn`, the `C_sign` and `C_setAttributeValue` functions, as they satisfy the aforementioned requirements and replaying them would not result in a logical error[5]. We conducted five individual rounds, in the following order:

1. Test the `C_logIn` function.
2. Test the `C_sign` function, without authentication. If not successful then,
3. Test the `C_sign` function, with the `C_logIn` function.
4. Test the `C_setAttributeValue`, without authentication. If not successful then,
5. Test the `C_setAttributeValue`, with the `C_logIn` function.

Vulnerabilities Found The smart-cards from which we were able to extract meaningful state-machines are Atos CardOS V4.3, Atos CardOS V4.4, Atos CardOS V5,

[3]An attacker posses the necessary credentials if he is able to perform cryptographic functions.

[4]A smart-card stores the handle as an attribute.

[5]For example, replaying the `C_generateKey` function would result to an error as that specific key has already be generated.

Safesite TCP IS V1. Our tests failed for Aladdin eToken Pro because the implementation requires a successful authentication for the initiation of a new function. The authentication mechanism being a challenge-response based on an agreed session key prevents replaying a sniffed session. Finally we were not able to test RSA SecurID and Athena ASEKey because the SmartCardLearner system cannot handle dongles (USB smart-cards).

Atos Cardos. The implementation of the Cardos V4.3, Cardos V4.4 and Cardos V5 smart-cards is identical. As such, the extracted state-machines and the vulnerabilities found are the same for all cards. The extracted state-machines are presented in Appendix B.1. A higher resolution of the diagrams is available at `http://bit.ly/2DIvSSo`. The results confirm that blind-replay attacks are feasible to these cards. Moreover, the results confirm that although the signature operation is protected by the user's PIN, the same session can be replayed and that the message to be signed, m, as well as the signature σ are not protected. Finally, the extracted state-machine of the `C_setAttributeValue` session indicates that changing a sensitive key's attribute can be performed without authentication. This is a new vulnerability that has not been previously discovered. Table 6.3 summarises the results of our tests.

Function	`C_logIn`	**Success**
`C_login`	PIN	✓
`C_sign`	✗	✗
`C_sign`	✓	✓
`C_setAttributeValue`	✗	✓

Table 6.3: Results obtained from the Cardos V4.3, Cardos V4.4 and Cardos V5 state-machines.

Safesite TCP IS V1. The extracted diagrams for the Safesite TCP IS V1 smart-card are presented in Appendix B.2. Both the `C_sign` and `C_setAttributeValue` functions require authentication. However, the implementation uses a trivial authentication mechanism (PIN), which allows eavesdropping and blind replay attacks. Finally, the message to be signed, m, as well as the signature σ are not protected. It is worth noting that the PIN verification can be achieved only if the dedicated file `a0000000180a0000016342` has been previously selected. Table 6.4 summarises the result of the state-machines analysis.

Function	C_logIn	**Success**
C_login	PIN	✓
C_sign	✗	✗
C_sign	✓	✓
C_setAttributeValue	✗	✗
C_setAttributeValue	✓	✓

Table 6.4: Results obtained from the Safesite TCP IS V1 state-machines.

6.4 Manual APDU-Layer Attacks

Automatically testing a smart-card for particular attacks, *e.g.*, setting a sensitive key as a non-sensitive by modifying the `CKA_SENSITIVE` attribute, requires the existence of a unified agreement between the smart-card vendors with regard to the implementation design. The purpose of the state-machines is to illustrate whether the smart-card is vulnerable to an umbrella of insecurities which can be then exploited to mount particular attacks. [Bozzato et al., 2016] demonstrated how it is possible to change particular attributes of a key although such action is forbidden by the PKCS#11 standard. In this section we present how we were able to perform the same set of attacks for the Cardos 5. Additionally we introduce a novel APDU attack that has not been previously discovered, the *redirection of keys*: tampering with the key's handle so that it points to another key. For example, assume a key sk_1 with the corresponding handle h_1; a redirection attack would be effective if h_1 points to sk_2, where $sk_1 \neq sk_2$. Such attack is particular useful when the attacker already knows, or is able to extract, the value of sk_2. Consequently, whenever the user requests any cryptographic operation with the key sk_1, the operation is performed with sk_2 which is controlled by the attacker.

Set of Attacks The attacks that we illustrate address the management of the key attributes. PKCS#11 [RSA Security INC, 2004] specifies the set of the attributes that can be set to a key, depending on the usage of the key. Given a sensitive key, our experiments are concerned with the following attribute violations: i) setting the `CKA_SENSITIVE` attribute from TRUE to FALSE, ii) setting the `CKA_EXTRACTABLE` attribute from FALSE to TRUE, iii) setting the `CKA_PRIVATE` attribute from TRUE to FALSE iv) altering the operations that the key can be used to *e.g.*, setting the `CKA_SIGN` attribute to a key that was not previously used for signing. Changing a key's attribute in one of the aforementioned ways is strictly forbidden by the API. However, as presented in

[Bozzato et al., 2016], we show that it is possible to change the attributes in that way through the APDU layer.

In addition to the attribute violation, we introduce the *redirection of keys*. We illustrate how an attacker can alter the handle of a sensitive key so that it points to a different key. The attack scenario which such violation could be helpful is when the attacker has imported his own key to the device. This can be achieved either from the API in a legitimate way (by invoking the PKCS#11 function), or in a malicious way either via the API or via the APDU layer (encrypt/unwrap attack).

Attacks in Practice We illustrate how we were able to perform the aforementioned attacks in the CardOS V5 smart-card. We chose that particular version as CardOS V4.3 has been studied before in [Bozzato et al., 2016]. We show that the same set of attacks is applicable to that version, and that the vendors have not fixed the issues discussed in [Bozzato et al., 2016].

The CardOS smart-cards store the attributes and the handle of a key in a single elementary file (EF) that consists of a single continuous sequence of records. For example, for the sensitive key with label *testkey* and id 12, the key's attributes can be accessed by first selecting the corresponding EF file:

```
00 a4 08 04 04 50154400.
```

and then by reading the 12th record of that EF:

```
00 B2 0C 00 EA
```

The contents of that record are:

```
0C 3B 30 39 30 10 0C 07 74 65 73 74 6B 65 79 03 02 06 C0 04 01 01 30
0E 04 01 67 03 02 02 64 03 02 03 B8 02 01 0C A0 02 30 00 A1 11 30 0F
30 06 04 04 50 72  4B 0C 02 02 04 00 02 01 11
```

The attributes that we are studying are the ones defined by the standard as *sticky on i.e.*, once set to TRUE they cannot be set to FALSE (more details discussed in Chapter 4). The goal of the defined tests is to check whether the value of such attributes can be changed at the APDU layer, while it is forbidden by the API. The attribute violations that we test are:

i) Change the value from TRUE to FALSE of the `CKA_SENSITIVE`, `CKA_ALWAYS_SENSITIVE`, and `CKA_PRIVATE` attributes, and change the value from FALSE to TRUE of the `CKA_EXTRACTABLE` and `CKA_NEVER_EXTRACTABLE` attributes. According to PKCS#11, these changes will lead to the exposure of the key's value.

ii) Change the value from FALSE to TRUE (and vice versa) of the `CKA_SIGN` and

attribute record	`0C3B303930100C07746573746B6579030206C0040101300E` `04017703020264030203B802010CA0023000A111300F3006` `040450724B0C02020400020111`
key count bytes	`0C`
key label bytes	746573746b6579
function bytes	0264
sensitive bytes	03B8
key handle bytes	50724B0C

Table 6.5: Representation of the *testkey* attributes.

`CKA_DECRYPT` attributes. Theoretically, this will enable/disable the usage of the corresponding operations to use that key.

iii) Alter the handle of the key so that it points to a different one.

The representation of the attributes is not standard. To identify the correspondence between the `CKA` attributes defined by PKCS#11 and their hexadecimal representation a manual analysis is necessary: comparison between the representation of different keys with the same attributes set to FALSE and to TRUE, as altering the attributes via API calls and tracking the changes in the attribute records. The deduced representation for the *testkey* key is presented in Table 6.5. The *key count bytes* hold the value of the key generation counter, the *key label bytes* hold the label of the key, the *function bytes*[6] specify which functions are permitted with that key, the *sensitive bytes*[7] specify the sensitive, private and extractable attributes, and the key handle bytes specify the handle of the key.

The attacks were achieved by

i) selecting the attribute file, *e.g.*, `00A4080404501544 00EE`

ii) updating the attributes record, *e.g.*, `00DC0E00`...

Results of the Attacks The conducted attacks indicate that the PKCS#11 attribute policy is only enforced at the API layer. In all cases mounting the attribute violations was successful, but it did not always result in a higher-level attack.

- *Exposure of the key value.* Theoretically, setting `CKA_SENSITIVE`, `CKA_ALWAYS_`

[6]Have to be decoded into binary.

[7]Have to be decoded into binary.

SENSITIVE and CKA_PRIVATE to FALSE as well as CKA_EXTRACTABLE and CKA_NEVER_ EXTRACTABLE to TRUE results in the exposure of the key value. This behaviour is not allowed by the Cardos API. However, the APDU layer does not comply with these restrictions, as we were able to change the attributes successfully. Table 6.6 summarises the attribute value changes (attack value) with regards to the value that was set by the API (original value).

Although the attack values appear at the API, the attack yields no security threat for the value of the key as it was not exposed. The attack takes effect on the security parameters of the key: after setting CKA_PRIVATE to FALSE authentication was no longer required. Moreover, depending on the combinations of the attributes, the key could no longer be used for sensitive operations such as the C_sign function.

attribute	**original values**	*attack*$_1$	*attack*$_2$
CKA_SENSITIVE	TRUE	FALSE	TRUE
CKA_ALWAYS_SENSITIVE	TRUE	FALSE	TRUE
CKA_EXTRACTABLE	FALSE	TRUE	FALSE
CKA_NEVER_EXTRACTABLE	FALSE	TRUE	FALSE
CKA_PRIVATE	TRUE	FALSE	FALSE
effect			
key value	✗	✗	✗
authentication protected	✓	✗	✗
sensitive operations	✓	✗	✓

Table 6.6: SENSITIVE, PRIVATE and EXTRACTABLE attribute value changes at the APDU layer.

- *Change of the key's function role.* The attributes of a key define the functions that are allowed to operate with it. Although these attributes are not *sticky on*, the standard suggests that the role of a key shall not change after its generation. The CardOs APIs do enforce that policy by treating the CKA_SIGN and CKA_DECRYPT attributes as *sticky on*. However, this restriction is not applied at the APDU layer. Our attack changed these attributes and we successfully performed the corresponding function. Table 6.7 summarises the results of the attacks.

attribute	**original template**	*attack*$_1$	*attack*$_2$	*attack*$_3$
`CKA_SIGN`	FALSE	TRUE	FALSE	TRUE
`CKA_DECRYPT`	FALSE	TRUE	TRUE	FALSE
effect				
`C_sign`	✗	✓	✗	✓
`C_decrypt`	✗	✓	✓	✗

Table 6.7: `CKA_SIGN` and `CKA_DECRYPT` attribute value changes at the APDU Layer.

- *Redirection of keys.* A key is referenced by its handle. According to the PKCS#11 standard, when a key is generated it is assigned a unique handle which must never change until the key is destroyed. Also, the private key handles must only be accessed if the user has previously logged-in. In the smart-card implementation, a key handle is implemented as a file path which points to the location of that key, and is stored as a key attribute. At the APDU layer, accessing the handle of the key is the same as accessing the key attributes. The CardOS smart-cards keep a counter c which tracks the number of the generated keys. For each freshly generated key, the card creates a new EF file. The path of that file is formed by c and serves as the key handle. For example the sensitive key sk_1, which was the first to be generated, has $c = 01$ and is stored at `50724B01`. The sensitive key sk_2, which was the second to be generated, has $c = 2$ and is stored at at `50724B02` *etc.* Given any two sensitive keys, sk_1 and sk_2, stored in the card, we have successfully changed the handle of sk_1 to point to sk_2. To evaluate the success of the attack we compared the signatures of the same message, σ_1 and σ_2, which were identical.

6.5 Summary

The APDU layer being complex and proprietary, in combination with the lack of well defined security requirements, makes reasoning about the underlying implementation a difficult task. The goal of this chapter is to provide a framework for addressing security, independently of the implementation.

We presented the first documented attempt towards a goal-driven threat model for the APDU layer with respect to PKCS#11. We addressed the security of that layer considering three abstractions: 1. the PKCS#11 standard, in which we consider vulnerabilities on the PKCS#11 specification as well as attacks that target the functions and objects defined by the standard, 2. the token operations, in which we address attacks that target the executed operations and the objects that are handled by the token, regardless of the higher-level standard that is implemented, and 3. the communication, in which we identify attacks that target the communication itself regardless of the nature of the participating parties (*i.e.*, smart-card, API) and the higher-level standard that is implemented. Not all attacks that target each abstraction overlap; by considering each abstraction individually we are able to capture attacks that otherwise would have been missed. We incorporated the STRIDE approach to model the potential threats as it addresses the basic security principles, CIA, and provides a goal oriented modelling technique. We narrowed-down the vulnerability search-space by identifying the core parts of the system communication that the identified attack scenarios exploit.

Using as a guide the set of possible vulnerabilities, we manually analysed the underlying security of the seven commercially available smart-cards that we had previously reverse-engineered with REPROVE. Additionally, we presented an ecosystem in which REPROVE interacts with SmartCardLearner to automatically extract meaningful state-machines of the APDU implementations. The extracted state-machines suggested both new and previously identified vulnerabilities. Finally, we showcased how it is possible to mount APDU attacks on the Atos CardOS smart-cards.

This chapter showed that proprietary implementations that are intentionally kept secret usually hide flaws and threats. Without extracting the implementation models such threats would have otherwise been unknown. One of the most surprising findings was to discover that even a respected smart-card vendor such as Gemalto[8], who is advertised for offering secure solutions, chose quick and cheap implementations over secure ones *i.e.*, computing cryptographic operation on the API instead on the card, and allowing secret key values to be exposed. Smart-card vendors advertise tamper-resistant products, capable of performing fast and efficiently the intended cryptographic operations. By not providing any implementation details, they request from their customers to blindly trust them. But as we presented, an analysis on such implementations indeed exposes major security issues. Execution of the intended cryptography on the API defeats the whole purpose of cryptographic smart-cards.

In general, a smart-card should offer the guarantees of producing the expected

[8]The analysed smart-card was initially offered by Aladdin. When Aladdin was acquired by Gemalto all products were offered by the later company. However as we show, Gemalto did not make any changes on the smart-card implementations.

outcomes. Even if the key values are not exposed, an adversary can still control the cryptographic computation outcome by tampering with the inputs and outputs. Therefore, security boils down to not only protecting the keys but also the computational mechanisms. To overcome such a problem, an implementation should:

1. Use an authentication mechanism that does not allow replaying; a PIN-only authentication cannot provide such a property. Incorporating a session key-agreement to the authentication can be one way around this.

2. Ensure that the transmitted data looks random to an adversary, to protect against eavesdroping and tampering. This property can be provided through secure messaging as specified in ISO 7816. Secure messaging allows different cryptographic operations over the whole command/response or part of it *e.g.*, encryption, MACing, hashes, digital signatures *etc.*

3. Enforce the PKCS#11 policies on the smart-cards; a secure PKCS#11 setting should ensure that not only the the API but also the smart-card adhere to the standard's specifications. This can be achieved by hard-coding the policies to the card's implementation. For example, to ensure that sensitive/secret keys are never exposed, the files that contain such keys must not provide read permissions; to avoid tampering a key's attributes in a malicious way, an update operation on the binaries/records that contain the corresponding attributes must not be permitted; to ensure that sensitive operations are initiated by legit users the file path of the operations must require authenticationt *etc.*

Part II

Bitcoin Smart-cards

Chapter 7

Introduction

Bitcoin is considered to be the most successful cryptocurrency to date, with its estimated average daily transaction value (30th November 2017) over US$2,9 million. As Bitcoin is becoming the most widely adopted digital currency and its number of users rapidly grows, many businesses choose Bitcoin for their transactions in order to reap the benefits of a larger user base. As more companies move into this space, each with its own solution, there is substantial resource and research investment into the security of the Bitcoin protocol and its transactions. Bitcoin is based on public key cryptography, which requires users to digitally sign their payments. Each account is defined by a private/public key pair, which is responsible for receiving and transferring funds. Payments are made into addresses that correspond to a hash of the public key of the recipient. Therefore, a salient aspect of Bitcoin is key management, as loss of the private keys effectively means loss of the funds. Keys are managed by the user's *wallet*, with two primary types of wallet existing in practice: online and offline wallets. Online wallets are the most accessible type of wallets and typically store the account's keys on a remote server. Remote storage, however, introduces trust and accessibility issues as the keys are no longer under the account holder's sole control and are only accessible if the server storing them is accessible. As a result, offline wallets have emerged as the strongest alternative of the two, at least for users that emphasise security. Even the strongest offline wallets, however, have their own shortcomings. In this part of the thesis we show how one can attack hardware-based offline wallets through reverse-engineering their low-level communication protocol. At the same time, we suggest ways to improve the said communication protocol to guard against such attacks.

Key Management and Bitcoin Wallets. The simplest approach to key management is to store the keys on a local disk, a solution proven to be vulnerable to dedicated malware [Litke and Stewart, 2014]. As a result, more generic methods have been introduced. Online-hosted wallets attract a large number of Bitcoin users, as they are offered as a service that is faster and safer than running the Bitcoin client locally. User accounts are hosted in remote servers and accessed through third-party Web services; wallets either store the keys also in remote servers, or locally in the user's web client (typically a web browser). The user accesses his wallet through web-based authentication mechanisms and all cryptographic operations take place server-side, typically in the Cloud. Although this approach is popular among Bitcoin users, certain security issues arise as the user's private keys can be exploited by the host. For instance, in 2013 the StrongCoin web-hosted wallet transferred an amount of bitcoins stored in their servers back to a different service, OzCoin, as it was claimed to be stolen [Bitcoinmagazine, 2013]. This transfer was done without any user consent. Online wallets are also common targets for *Distributed Denial of Service* (DDoS) attacks, *e.g.*, BitGo and `blockchain.info` in June 2016. Such examples raised concerns about the reliability of such wallets and fueled the trend for *cold storage*, and in particular *cryptographic tokens*. These devices not only store the keys securely, but also compute cryptographic operations in a tamper-resistant environment. Thus, most major companies have integrated their software wallets with hardware devices.

Hardware wallets aim to offer a secure environment to store sensitive keys and sign transactions. Hardware wallets typically implement their own API to communicate with Bitcoin. When a user requests a payment, the wallet's API creates the corresponding raw Bitcoin transaction and sends it to the hardware to be signed. The hardware signs the transaction and returns the signature and the corresponding public key to the API, which will then push it to the network. The Bitcoin wallets currently on the market incorporate either microcontrollers or smart-cards. As of February 2017, the hardware wallet options suggested by `bitcoin.org` are the LEDGER wallets, which are based on smart-cards, or the *Trezor*, *Digital Bitbox* and *Keepkey* wallets, which are based on microcontrollers. All wallets offer two versions: (*a*) a plain USB dongle, or (*b*) a USB device with an embedded screen for the user to verify the transaction. Regardless of the version, the hardware device is responsible for managing all keys and for signing the transactions. The trade-off between versions is in terms of security, price and usability. While devices with an embedded screen provide more guarantees that the transactions are not tampered with, they also delay transactions as they require constant user interaction to verify every step; additionally, they are more expensive. As of February 2016, the LEDGER *HW1 wallet*, a plain USB dongle, costs around US$20.

The *Trezor* wallet, which has a secure screen, costs around US$119.

Offering a tamper-resilient cryptographic memory is not enough on its own to guarantee against transaction attacks. Unauthorised access to the signing oracle of the wallet is not very different from plain access to the keys themselves, as both allow the funds to be stolen. Processing a Bitcoin request involves communication between the hardware wallet and third-party systems. The lack of a general threat model for the Bitcoin wallets and well-defined specifications of that communication leads to proprietary implementations. As previous studies on different protocols have shown (*e.g.*, [Bozzato et al., 2016, Gkaniatsou et al., 2015, De Koning Gans and de Ruiter, 2012]), such practice often results in insecure low-level implementations that are prone to *Man-in-the-Middle* (MitM) attacks.

In this part of the thesis we stress the importance of securing the low-level communication of Bitcoin hardware wallets. We show that by taking advantage of that communication layer it is possible to propagate the attacks directly to the underlying Bitcoin transactions. The attacks we address are general and target any low-level communication with hardware wallets. Applying them in practice is a matter of adapting them to the corresponding hardware implementation. The security of microcontrollers has been extensively examined, and a number of fault and side-channel attacks have been found, *e.g.*, [Kocher, 1996, Kocher et al., 1999, Biham and Shamir, 1997, Genkin et al., 2014]. Therefore, we focus on smart-card based wallets, which provide guarantees against physical and interdiction attacks and have traditionally been used for key management and cryptographic operations. As of April 2017, LEDGER is the only company offering smart-card solutions. The LEDGER wallets are EAL5+ certified and are advertised as the most secure, tamper-proof and trustworthy devices for managing Bitcoin transactions.[1] All LEDGER wallets implement key derivation based on the *Bitcoin Improvement Proposal 32* (BIP32) [Wuille, 2017]. LEDGER wallets are either plain USB dongles or have an embedded screen. In both cases they communication with the API via the APDU layer.

We only consider client-side security rather than security in the Bitcoin network, although a single wallet attack may escalate. General attacks on Bitcoin wallets that could be applied to several users simultaneously are a cheap, easy and efficient way to gain access to multiple accounts. The LEDGER wallet API is open-source[2] and can

[1]See `http://bit.ly/2rcBsuw`, `http://bit.ly/2B4Cg4f`, `http://bit.ly/2DmWKKG`.

[2]https://github.com/LedgerHQ/ledger-wallet-chrome

be downloaded from *Chrome Web Store*[3]. As such, massively attacking the LEDGER wallets can be achieved by a possible malware download from Chrome Web Store.

Contributions and Roadmap. To the best of our knowledge our work is the first to: (i) stress the importance of securing Bitcoin transactions and preserving the account's privacy at the wallet level, (ii) consider a minimal threat model for hardware Bitcoin wallets, and (iii) address the security issues originating in low-level communication of Bitcoin devices, by showcasing practical attacks.

- Chapter 8 provides the background and the context of our work. We provide an overview of the Bitcoin protocol and its specifications. We review the current state-of-the-art at Bitcoin hardware wallets and show that the undeline protocol they implement is the same. The idea of attacking Bitcoin hardware wallets is novel and there does not exist in the literature work close to ours. However, we discuss the major Bitcoin vulnerabilities that have been discovered.

- As, the LEDGER protocols are not publicly available, we reverse-engineered the communication protocol and abstracted its implementations. Chapter 9 presents the LEDGER protocols.

- In Chapter 10 we articulate a general purpose threat model for Bitcoin wallets and show how we have successfully mounted the identified attacks on LEDGER wallets. To that end, in Chapter 10 we propose a lightweight and easily adaptable fix that requires minimal changes.

[3]https://chrome.google.com/webstore/detail/ledger-wallet-bitcoin/kkdpmhnladdopljabkgpacgpliggeeaf

Chapter 8

Background

Bitcoin is a *Peer-to-Peer* (P2P) payment system that utilises public-key cryptography and consists of addresses, transactions and blocks. Transactions define the operations that combine the transferred funds and blocks keep records of the validated transactions. Transactions are publicly available as they are broadcast through the network to all Bitcoin nodes. A node is any computer that connects to the Bitcoin network. When a transaction is sent to the network, it will be put into a block to be verified by the *miners*. The miners are individuals who are responsible for processing the transactions by computing a proof of work (POW), which consists of solving a hard cryptographic problem, to be able to add a new valid block to the public ledger. Once a transaction is successfully verified it is hashed[1] and stored in a public ledger, the block chain. Then, the funds that this transaction transfers can be spent on a new transaction, creating a link between them.

Transactions: An account is defined by one or more public/private key ECDSA keypairs. The 160-bit hash of each public key corresponds to a unique address in which the account may accept funds. For an account to spend a specific amount of Bitcoins the addresses belonging to that account must have at least an equal amount of funds. In each transaction the available funds, represented by inputs, and the payment details, represented by outputs, are specified. Each transaction is uniquely identified by *txid*, a unique id. *txid* is the double-SHA256 hash of the entire transaction block. A transaction may have multiple inputs and outputs. The input funds belonging to a public key can only be spent by a proof of ownership: the transaction must be signed with the corresponding private key. The signature and the public key are then sent to the network for verification. Upon successful validation the funds are transferred to

[1]That hash corresponds to a unique transactions identifier (txid).

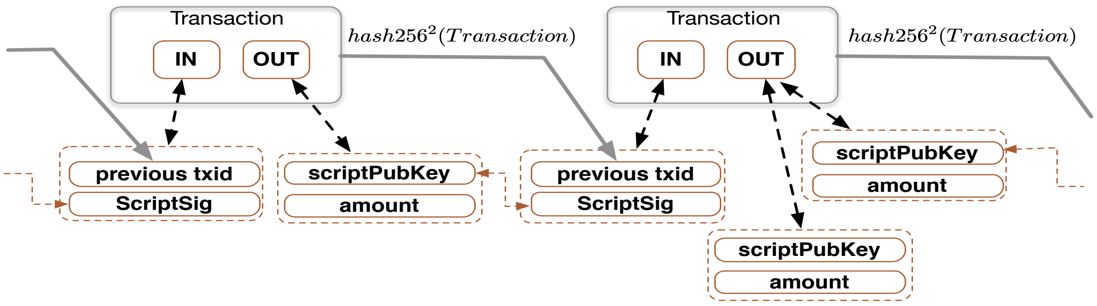


Figure 8.1: *Transactions on the public ledger*: the outputs of a past transaction are the inputs of a future one. A transactions is referenced by the corresponding txid which is the double SHA256 hash of the whole transaction block. The output of a transaction can be spent only by the owner of the keypair whose public key is defined at the ScriptPubKey field. The spender proves the ownership by signing a payment using the corresponding private key and including the payment to the ScripSig field.

the destination addresses specified at the outputs.

More formally, let *Bob* be a user with a private/public key pair (sk_b, pk_b) and x_b be the address of that user, where $x_b = hash(pk_b)$; let $y_x = hash(t_x)$ be the hash of transaction t_x that transferred x funds to the x_b address (inputs). The transaction that transfers x funds to Alice's address z_a (outputs) is the signature σ_{sk_b} of y_b, x and z_a using private key sk_b: $sign(y_b, x, z_a)_{sk_b} \rightarrow \sigma_{sk_b}$.

Once a transaction is formed, it is broadcast to the network along with the corresponding public key where it will be validated by the miners and then added to the public ledger. The public ledger, also known as block chain, consists of chains of blocks that describe past transactions. The purpose of the block chain is to confirm transactions that have taken place, to the rest of the network, and to offer a way for distinguishing legitimate transactions. A transaction is added to the block chain if: (a) the outputs of the transaction do not exceed the inputs, and (b) the verification of the signature, is successful. The miners that are responsible for that process are paid a fee for each transaction processing: the amount remaining when the output value is subtracted from the input constitutes the transaction fees. By default Bitcoin uses floating fees and depends on the transaction size.

Bitcoin Transactions in Practice Transactions in Bitcoin are expressed in a stack-based language known as the *Bitcoin raw* protocol or *the Script language*, which defines the conditions of the inputs and the outputs. The language has 80 different single-byte opcodes and includes operators to express operations such as arithmetic, bitwise, string conditionals and stack manipulation. It also includes cryptographic operations such as SHA256 for hashing and CHECKSIG for verifying the signature of a message with a public key. According to [The Bitcoin Wiki, 2014b], a transaction is defined in blocks of bytes, in which the inputs and the outputs are defined. In each block the inputs of that transaction and the corresponding outputs are defined following a specific structure. Table 8.1 presents the specific structure of a transaction block and the abbreviations that we will use in the next chapters. Apart from the more technical fields that a transaction block should define *e.g.*, the block format version or the block lock time *bt*, the core of the block consists of 1. *txid* which defines from which transaction the input funds come from, 2. *scriptSig* which is the signature of the current payment with *txid* and consequently proof of ownership, and 3. $addr_p$ which defines the destination of the outputs.

The way the transaction blocks are structured presents how the transactions are connected in the public ledger, as shown in Figure 8.1. Including the *txid* of the input funds imposes a chronological order of the transaction. The outputs of a transaction will be the inputs of another transaction which has ownership over *scriptPubKey* and proves that ownership with *scriptSig*.

Transactions including Segregated Witness: The standard structure of a transaction block, as presented in Table 8.1, does not include the amount of the input funds; only the *txid* of the previous transaction is provided and the signature with the key which unlocks the outputs of *txid*. When the new transaction is sent to the network for confirmation, the miners will unlock all funds in *txid* that belong to the signature verification regardless of what is currently spent. If a payment only uses part of these funds, then the wallet creates a new public key in which the remaining balance minus the transaction fees are sent. Thereby, it is essential for the wallet to have access to the inputs amount even if it is not declared within a transaction block.

The limitation of storage capabilities in hardware wallets makes it impossible for them to save data about previous transactions and, consequently, keep track of the funds flow. Attacks on the input funds are limited, as the miners check the amount that it is unlocked from the previous blocks. However, lack of knowledge of the amount of the input funds creates space for the already known Attack 1 in which the attacker decreases the declared amount of the input funds.[2]

Attack 1. *Given a previous transaction* $txid_i$ *which transfers* $amount_p$ *to* $addr_p$*, the*

[2]http://bit.ly/2C0T1hJ

	v: version	4 bytes
inputs	ic: input count	1 byte
	$txid_i$: previous transaction id (hash)	variable length
	pc: previous output index	4 bytes
	$sigL$: script signature length	1 byte
	$scriptSig$: script signature	variable length
	s: sequence	4 bytes
outputs	oc: output count	1 byte
	$amount_t$: value	8 bytes
	l: script length	1 byte
	$addr_p$: scriptPubKey	variable length
	bt: block lock time	4 bytes

Table 8.1: The structure of a transaction block. v is a fixed constant that defines the block format version. ic is a counter for the inputs. $txid_i$ is the reference to the previous transaction whose outputs will fund the current transaction. pc is a reference to the outputs of $txid_i$ that will be used. $sigL$ is the length of the signature. $scriptSig$ is the signature of the current transaction with the private key that correspond to the previous transaction outputs. s is a fixed constant that defines the end of the inputs declaration. oc is a counter for the outputs of the current transaction. $amount_t$ corresponds to the amount to be spent and l to the length of the destination public key. $addr_p$ is the public key in which $amount_t$ will be sent.

owner of $addr_p$ proceeds to a payment $amount'_p$ with fees f. The wallet calculates the remaining $balance = amount_p - (amount'_p + f)$ and transfers balance to a new key belonging to that account. By changing the original input funds to $amount_a$ so that $amount_p > amount_a$ the wallet will calculate the wrong balance, $balance_a$ according to $amount_a$. When that transaction is received by a miner for validation, the original funds, $amount_p$ from txid will be unlocked and the miner will check the outputs based on $amount_p$. The difference will then be kept by the miner leading to an inflated fee.

One way for hardware wallets to overcome this issue would be to recalculate the previous transaction's hash *txid*, by receiving the entire transaction block, and then compare it with the one sent to be included during the construction of the current transaction. If the input funds are tampered, then the transaction hash will not be the same and the hardware will abort the process. Although this approach seems to offer a good solution, it is not optimal when the hardware has to deal with a large number of inputs, as it will cause great delays to the payment process.

Segregated Witness [Bitcoin, 2016] (SegWit) solves this problem and is currently the approach adopted by all hardware wallets as it allows hardware wallets to avoid that time consuming verification. With SegWit the hardware no longer requires to process the whole previous block, as transactions now include an extra field in which the amount of the inputs is specified. The hardware wallets first hash the inputs and then sign that hash. As the amount is now included in the signature, Attack 1 is not possible.

LEDGER incorporates a proprietary SegWit by enforcing the API to send a detailed description of the inputs before the payment processing: the API forms a pseudo transaction block which only has the inputs, and sends it to the dongle, through a set of `trusted_input` commands. The dongle parses the block (bytewise concatenation $\oplus$) and returns its signature Sig_i. When the API creates the actual transaction each Sig_i defines the corresponding input.

Importance of Privacy in Bitcoin. As in all payment systems, practical privacy is an important aspect in Bitcoin [Nakamoto, 2008]. Bitcoin is a transparent payment system that enforces all transactions to be permanently posted on a shared public ledger, on which the entire Bitcoin network relies. Blockchain offers a decentralised ledger making it possible to facilitate transactions without the need of trusted third parties. In such a system, strong privacy is difficult to preserve; the transactions and the balance of any address are publicly available. However, using pseudonyms to determine the account details is one of the ways that Bitcoin may provide unlinkability. Additionally, Bitcoin Core claims[3] to provide strong privacy for the sent transactions by relaying all transactions that are sent to the network. As such, tracking the transactions becomes extremely hard. Perfect privacy for received transactions is another aspect that Bitcoin Core addresses, claiming to give information theoretic (perfect) privacy[4].

Apart from the privacy preserving protection actions taken in Bitcoin Core, the final outcome boils down to the wallet implementation. Avoiding reuse of the same address for different transactions has been suggested as a good policy for keeping the account unlinkable[5]. Even if an attacker knows a set of addresses and their balance, he cannot link them to the same account. Such a tactic has been adopted by almost all

[3]www.bitcoin.org/en/bitcoin-core/features/privacy

[4]The privacy cannot be broken even if the attacker has unlimited computing power

[5]https://bitcoin.org/en/protect-your-privacy

available wallets. Each time the account needs a receive address, the wallet generates a fresh public key pk_i and obtains the address $x_i = hash(pk_i)$.

To conclude, although Bitcoin is a transparent system, privacy preserving measures are taken both at the network and at the wallet layers. The core idea for keeping an account unlinkable is using distinct addresses for each transaction, implying that the account's public keys should remain private. As such, the wallet must not leak any information regarding the public keys it manages.

8.1 Bitcoin Wallets

Hardware wallets aim at eliminating the API access to the wallet's keys and offer an isolated environment for the cryptographic operations. The main responsibilities of such devices are the wallet's key management and the transaction signatures. Currently, hardware wallets do not have access to the network; thus, the transactions are pushed by their APIs. When a device is requested to sign a transaction, in addition to the outputs it also requires access to the available inputs. Based on the Bitcoin protocol the inputs are specified according to the previous transactions whose outputs transferred funds to the keys that belong to the wallet. However, hardware wallets cannot save them internally due to memory limitations. Thus, a communication with the API is required each time the hardware wallet is requested to process a transaction in which the inputs, outputs and the transaction signature are exchanged. Currently, apart from the Digital BitBox wallet, none of the wallets use an encrypted channel for that communication.[6]

A generic protocol that captures the process of a payment processing in hardware wallets is depicted in Figure 8.2. The API broadcasts the input funds to the device (`tx_input`) and the payment details (`tx_output`) and requests the transaction signature (`sign_tx`). Then, if the device supports a second factor verification mechanism for the payment, the device will sign the transaction only after the user's approval. If the device does not support such a mechanism, it will sign it immediately.

Although not all devices include a verification of the payment by the user, those that do adopt a different mechanism depending on the hardware specifications. The plain USB stick wallets that do offer a second factor verification usually incorporate a security card that is used to verify the payment address, or require pairing with a mobile application from which the user approves the payment. The wallets with an

[6]As stated on each wallet's website.

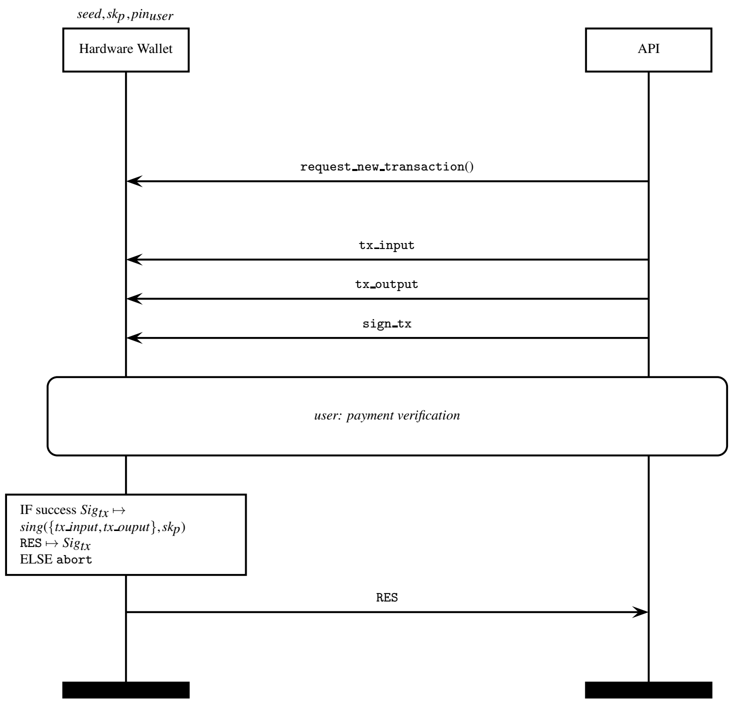

seed, skp, pin_user
Hardware Wallet
API
request_new_transaction()
tx_input
tx_output
sign_tx
user: payment verification
IF success Sig_tx ↦ sing({tx_input, tx_ouput}, skp) RES ↦ Sig_tx ELSE abort
RES

Figure 8.2: General payment protocol for hardware wallets

embedded screen usually present the payment data to the user and expect the user to verify it directly to the device *e.g.*, by pressing a button.

As of February 2017, the hardware wallet options suggested by `bitcoin.org` are the three LEDGER wallets, which are based on smart-cards; or *Trezor*, *Digital Bitbox* and *Keepkey*, which are based on microcontrollers. All wallets offer two versions: (*a*) a plain USB dongle, or (*b*) a USB Human Interface Device (HID) with an embedded screen for the user to verify and confirm the transaction. All these are USB devices that implement a Hierarchical Deterministic Wallet. Table 8.2 summarises the characteristics of each wallet. Depending on the secure element a wallet incorporates, it follows a different communication protocol. Currently, apart from Digital BitBox, none of the wallets uses a secure communication channel.

wallet	secure element	HID	payment verification	channel	encrypted channel
LEDGER `HW.1`	smart card	×	×	APDU	×
LEDGER `Nano`	smart card	×	✓	APDU	×
LEDGER `Nano S`	smart card	✓	✓	APDU	×
`Trezor`	microcontroller	✓	✓	protocol buffer	×
`KeepKey`	microcontroller	✓	✓	protocol buffer	×
`Digital BitBox`	microcontroller	×	✓	HID Api	✓

Table 8.2: Bitcoin hardware wallets characteristics

USB HID microcontroller Wallets. USB Human Interface Devices (USB HID class) are USB wallets with interfaces, such as a screen and buttons, to allow the user interaction directly with the device. These devices have an embedded microcontroller that performs the wallet operations. The companies that offer such wallets are Trezor, Keepkey and Digital BitBox. The API-dongle communication is bidirectional, and is achieved via the protocol buffers [Google,] mechanism which aims at serialising structured data. The Trezor wallets use a binary message format consisting of a header followed by a detailed section. For example, the API will send the input details by sending the message:

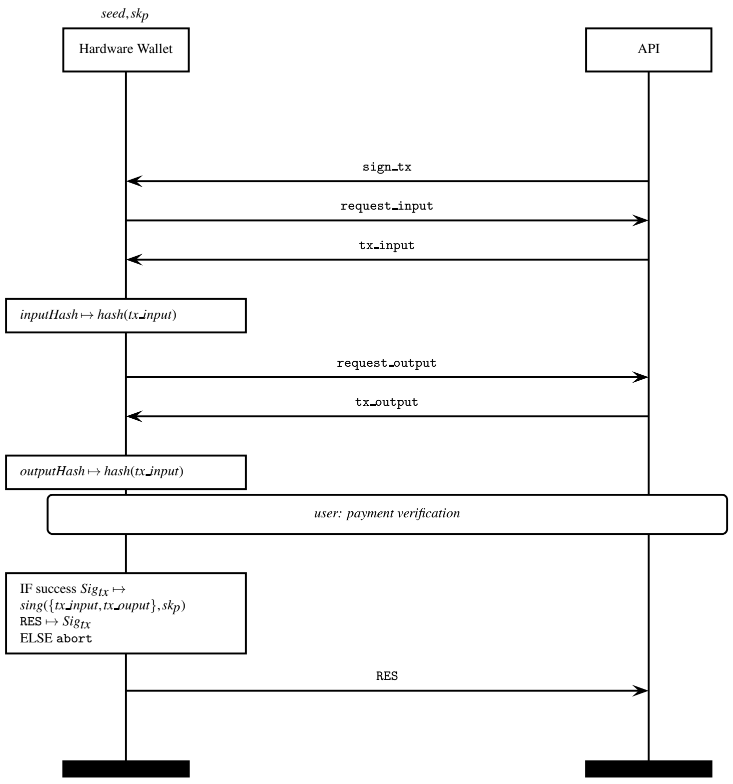
```
{ Message:TxAck
tx_=" "
bin_outputs_count: " "
inputs_count:" "
prev_hash: " "
prev_index: " "
script_type: " "
script_sig: " "
outputs_cnt: " "
outputs_count: " "
lock_time: " "
version: " " }
```

following the structure as presented in Table 8.1. The low-level payment protocol for the Trezor and KeepKey devices is depicted in Figure 8.3 and proceeds in an unencrypted channel while the communication in Digital BitBox is encrypted. All three devices require the user's confirmation to proceed to the signature.

Smart-Card Wallets. Smart-card wallets are plain USB or USB HID (Human Interface Devices) devices that have an embedded microprocessor. Currently, the only smart-card wallets are the Ledger HW1 and Ledger Nano which are plain USB and Ledger Nano S which is USB HID. The API-dongle communication is achieved through the APDU layer and consists of command-response pairs. For example, the API will send the outputs of a transaction, with the following command:

Cla	Ins	P1	P2	Lc	Data
e0	46	02	00	48	address, ammount, fees

A response consists of an optional body, the response data, and a compulsory 2-byte trailer of bytes SW1 and SW2 encoding the expected status of the card after processing



$seed, sk_P$
Hardware Wallet
API
sign_tx
request_input
tx_input
$inputHash \mapsto hash(tx\_input)$
request_output
tx_output
$outputHash \mapsto hash(tx\_input)$
user: payment verification
IF success $Sig_{tx} \mapsto sing(\{tx\_input, tx\_ouput\}, sk_P)$
RES $\mapsto Sig_{tx}$
ELSE abort
RES

Figure 8.3: Payment protocol for trezor and KeepKey

the command).

8.2 Hierarchical Deterministic Wallet

Bitcoin does not follow a specific protocol to derive keys. There are different approaches, with the simplest one being each account holding a single $\{sk, pk\}$ pair and consequently a single receive address. Such practice is not strongly advised as reusing the same address causes account linkability problems. Other practices involve a password-based key derivation, *e.g.*, [RSA Laboratories, 2012] proposes using the PBKDF2 function to derive a single keypair; a new password has to be used for another pair to be generated. Currently all hardware wallets implement a *Hierarchical Deterministic* (HD) wallet of BIP32, which generates a new key-pair for each address request [Wuille, 2017].

HD wallets derive fresh private keys from a common master key. Such a feature makes it easier to back up the wallet as the user only needs the master private key. For the creation of a new wallet a 128- to 512-bit seed s, a sequence of random numbers, is generated. Then, a checksum of that sequence is created by taking the first bits of its

SHA256 hash and adding it to the end of the random sequence. The result is divided into sections of 11 bits which will be the indexes of a 2048 predefined dictionary of words, and a mnemonic phrase of 12 to 24 words is generated. All keys that will be generated later will be derived from s, making that seed or the mnemonic phrase enough for a user to restore the wallet. The master private key sk_m is generated by a function $sk_m = g(hash(s))$ where g is a function that splits $hash(s)$ into two 32-byte sequences: L being the master private key sk_m and R being the master chain node.

Given a master key pair (sk_m, pk_m), an HD wallet generates and maintains a sequence of child private $sk_i, sk_2, ...$ and public $pk_1, pk_2, ..$ keys from the master private key sk_m. A key sk_i is derived by the functions
$sk_i = sk_m + hash(i, pk_m)\ (mod\ N)$,
$pk_i = pk_m + hash(i, pk_m)N$ or equally sk_iN
where i denotes the index of the key, and $hash$ is the HMAC-SHA512 function. Child public keys pk_i can be derived only by knowing the master public key pk_m and the index i.

Apart from generating fresh keys for each receive and change address, HD wallets are particularly attractive to large organisations due to the hierarchy of keys offered: each department can have its own key-pair, which is the child of a single root.

Limitations of HD Wallets. HD wallets allow the generation of child keys from their parent keys: a set of child public keys $pk_{i+1}, pk_{i+2}, ..$ can be generated by their parent key pk_i, a set of child sensitive keys $sk_{i+1}, sk_{i+2}, ..$ can be generated by their parent sk_i. Although this approach is practical for minimising access to the account's sensitive keys, or the management of different pairs by different wallets, the key dependence can cause security implications.

Access to the parent key implies access to all child keys. Having access to pk_i enables the generation of all child keys and consequently the ability to track the flow of the funds of a given account. The same vulnerability applies to the sensitive keys, as knowledge of the parent key sk_i enables the generation of the whole sub-tree of sensitive keys.

A more sophisticated, but known (also stated in [Wuille, 2017]) vulnerability of HD wallets is the extraction of a parent private key sk_i. If an attacker has access to the index of that key i, the corresponding public key pk_i and a random child private key sk_j, where $i < j$, the parent private key can be retrieved by applying $sk_i = sk_j - hash(j, pk_i)$ $(mode\ n)$. Thus, leakage of any of the child private keys also compromises the master private key.

Limitations of HD Hardware wallets. Whether or not a hardware wallet is vulnerable to the aforementioned limitations of HD wallets, boils down to the security of their key management. Although none of the devices exposes directly a private key, almost all the devices expose their public keys. Table 8.3 presents how hardware wallets with known specifications,[7] manage their public keys. Although in all cases pk_m is generated on the device, it is then sent to the API in plaintext. In all wallets the receiving keys pk_i are generated API-side.

Transmission of pk_m in the clear makes the wallets vulnerable to eavesdropping and thereby to privacy loss. Such behaviour is vulnerable to sniffing attacks, exposing the wallet vulnerable to privacy loss and linkability issues: possession of pk_m entitles possession of all public keys that belong to a wallet, making all the wallet's transactions traceable and linkable to that specific wallet. Furthermore, the public keys that correspond to the receiving address are generated in all wallets API-side meaning that the devices only manage the corresponding private keys. Although this behaviour seems reasonable, APIs do not offer a tamper resistant environment, making the wallets vulnerable to an injection attack: the attacker imports his own public keys so that any future receiving funds are sent to his account.

wallet	pk_m **generation**	pk_m **sent to API**	pk_i **generation**
LEDGER `HW.1`	on-device	✓plaintex	API
LEDGER `Nano`	on-device	✓plaintex	API
LEDGER `Nano S`	on-device	✓plaintex	API
`Trezor`	on-device	✓plaintex	API
`KeepKey`	on-device	✓plaintex	API

Table 8.3: Public-key management in hardware wallets.

8.3 Related Work

Previous work on attacking Bitcoin has exposed malleability attacks, where the the adversary forces the victims to generate a transaction to an address controlled by him. When a victim broadcasts the transaction to the network, the adversary obtains a copy of that transaction that he modifies by tampering the signature without invalidating it. That modification results in a different transaction identifier (hash). The adversary

[7]We only show the devices that we reverse-engineered along with the ones whose specifications are known. The specifications of DigitalBitbox are not publicly available.

then broadcasts the tampered transaction to the network, resulting in the same transaction being in the network under two different hashes. As a single transaction can only be confirmed once, only one of these two transactions will be included in a block and the other will be ignored. The attack is successful if the attacker's modified version is accepted. Although this attack is not new, it was given great attention after the malleability attack in MtGox [Decker and Wattenhofer, 2014], the first and one of the largest Bitcoin exchanges, in 2014. Since then different malleability attacks and solutions have been proposed, *e.g.*, [Wuille, 2014, Decker and Wattenhofer, 2014].

Double spending is a class of attacks on Bitcoin transactions, where the user spends the same coin twice. The feasibility of double spending attacks by using hashrate-based attack models was studied in [Nakamoto, 2008, Rosenfeld, 2014]. It was shown that the attack is successful whenever the number of confirmations of a dishonest transaction is greater than the number of confirmations of the honest one. The work presented in [Karame et al., 2012] proposes the exploitation of non-confirmed transactions to implement double spending attacks on fast payments, and [Rosenfeld, 2014] shows how such attacks coupled with high computational resources can have a higher success rate.

Bitcoin provides a limited form of unlinkability as users can generate new addresses at any time. Though privacy is a concern of the original specification [Nakamoto, 2008] the public nature of Bitcoin renders strong privacy difficult to achieve. For instance, by tracing the flow of coins it is possible to identify their owner [Herrera-Joancomartí, 2014]. Likewise, in [Androulaki et al., 2013] the authors study how transaction behaviour can be linked with a single account.

The aforementioned attacks target the network layer and assume that the wallets processing the transactions are trustworthy. As many malware attacks have gained publicity *e.g.*, [Tribbleagency.com, 2012, Poulsen, 2011, Huang et al., 2014] or the malware attack on the Bitstamp wallet that costed US$5M [Higgins, 2015], the importance of protecting Bitcoin wallets has been repeatedly stressed [The Bitcoin Wiki, 2014a]. In [Barber et al., 2012] the authors propose as a solution to malware a *super-wallet* in which the funds are split across multiple devices using cryptographic threshold techniques. The importance of ensuring that a wallet is secure is also presented in [Turuani et al., 2016] where the authors formally analyse the authentication properties of the *Electrum* wallet. The authors of [Lim et al., 2014] and [Bamert et al., 2014] argue that Bitcoin wallets be tamper-resistant and propose cryptographic tokens as a countermeasure to malware attacks. Our work exploits Bitcoin transactions at the wallet level. Instead of attacking the Bitcoin raw protocol directly, we show the importance of the protocols connected to the Bitcoin implementations. Attacking such protocols overrides any security restrictions that expensive hardware additions may add, and can be equally harmful to attacking the Bitcoin raw protocol itself.

Chapter 9

Extracted Ledger Wallet Protocols

The low-level communication layer of LEDGER wallets, defined by the APDU layer, is crafted to implement the Bitcoin raw protocol. The communication consists of a series of raw hexadecimal command-response pairs between the API and the hardware: the API retrieves data or requests the hardware to execute a specific operation via APDU commands; whereas the hardware responds to that request via APDU responses. For example, in the following sequence:

command	`e04800001f058000002c80000000800000008000000000000000000000000c04040606020000` `000001`
response	`3044022033128d0d576487e2e0c5892c0915564a6a5f119e698c033262d6605279` `43a16d022009caa037703d9a3dbf7eec4cecca08bf33b3b9a18ef929a810f8faf6` `ab0f1c7a01`

command retrieves the signature (*response*) over some transaction data.

The LEDGER communication protocols are closed-source and there does not exist any public information on how the Bitcoin specifications are translated into the APDU layer. A large part of our work has been to reverse-engineer the APDU layer and extract the implemented protocol. This was achieved by creating a man-in-the-middle sniffer sitting on top of the Ledger API, capable of recording and interfering with the communication during any active sessions with the dongle. To abstract the protocol from the actual implementation and to infer the dongle's operations we ran a series of sessions on three different Nano dongles and one Nano S[1], compared the APDU command-response pairs, analysed the exchanged data and mapped it to the Bitcoin

[1]The protocol of Nano S is very similar to that of Nano, thus it was not necessary to test it on a different dongle.

raw protocol. We concluded that during an active session four protocols may be executed:

(a) *Dongle Alive*: the initial communication when the dongle is plugged-in.

(b) *Setup*: wallet configuration and generation of the master keypair $\{sk_m,pk_m\}$.

(c) *Login*: user authentication to the dongle, and *vice versa.*

(d) *Payment*: processing of a payment transaction.

The *Dongle Alive* and *Login* protocols run once each time the dongle is connected to an active API. The *Payment* protocol repeats each time the user requests a payment. To proceed to a payment the user is not required to re-authenticate. The *Setup* protocol is executed once for initialising the wallet and each time an account restore is required; user authentication is its prerequisite. The dongle communicates with the API only when one of the four protocols are executed or when a firmware update is requested.

Commands used during the communication. The wallet communication consists of hexadedimal messages between the API and the dongle. To make the analysis readable we present the command-response messages in the form of $\mathtt{c}(p_i,p_{i+1},\ldots,p_n) \rightarrow r_1,r_2,\ldots,r_m$, which denotes that the API sends the command `c` with parameters $p_i,p_{i+1},\ldots,p_n$, $i \geq n$, $n \geq 0$, to the dongle; and the dongle responds with $r_1,r_2,\ldots,r_m,m \geq 0$. If $m = 0$ the dongle either replies with `OK` (success) or `error` (failure). Table 9.1 lists the communication primitives used to describe the protocols.

Keys that appear during the communication. Our analysis showed that LEDGER wallets manage the following key types:

(*i*) $\{sk_{att},pk_{att}\}$: predefined attestation keys, used for the dongle's firmware authentication and for setting up third-party hardware,

(*ii*) $\{sk_m,pk_m\}$: the master keypair from which all keys are derived,

(*iii*) $\{sk_i,pk_i\}$ pairs: transaction related keys, *i.e.*, keys$\{sk_r,pk_r\}$ for receiving funds and $\{sk_c,pk_c\}$ for transferring the change of a transaction. All keys, besides pk_r, are generated and stored dongle-side.

(*iv*) pk_{kp}: a symmetric key for the encryption/decryption of the wallet's key-pool. As most Bitcoin wallets do, LEDGER software maintains a key-pool of 100 randomly generated addresses: each time the wallet requires a new address it picks one from the key-pool which is then refilled. Based on the original Bitcoin client (*i.e.*, the Satoshi client) the key-pool gets encrypted (AES-256-CBC) with an entirely random master key [The Bitcoin Wiki, 2014c]. This master key is encrypted with AES-256-CBC with another key derived from a SHA-512-hashed passphrase. In the original implementa-

command	**meaning**
`get_firmware_version`() $\rightarrow fV$	returns the dongle's firmware version *fv*
`get_wallet_public_key`($bipDer_i$, $findex_i$, $lindex_i$) $\rightarrow pk_i$	given the number of bip derivations $bipDer_i$, the first index $findex_i$, the last index $lindex_i$, returns the public key pk_i
`get_device_attestation`(*blob*) $\rightarrow \{Sig_{att}, attId, attDer, frwVer, modes, currentMode\}$	returns the signature Sig_{att} of *blob* which is the concatenated byte-string of firmware version *frmwVer*, with the private key sk_{att} the verification key parameters *attId*, *attDer*, the operation modes *modes*, the current mode *currentMode* and *frmwVer*
`verify`(*pin*) $\rightarrow$ `OK`	sends the user's *pin* to the dongle; if correct, the dongle replies `OK`
`set_operation_mode`(*secFac*, *opMode*)$\rightarrow$ `OK`	sets the second factor authentication *secFac* to true/false and the wallet operation mode *opMode* to standard/relax/developer
`sign`($bipDer_i$, $findex_i$, $lindex_i$, *m*) $\rightarrow$ `OK`	initialises the signature of the message *m* with the private key sk_i that corresponds to ($bipDer_i$, $findex_i$, $lindex_i$)
`sign`(*pin*) $\rightarrow Sig_m$	returns the signature Sig_m of message *m* with key the private key sk_i if the *pin* it provides is correct
`setup`(*pin*, *seed*, *genKey*) $\rightarrow$ `OK`	sets up a new user's *pin*, stores a new *seed* and requests from the dongle to generate, *genKey*, a new $3DES_2$ key
`set_keyboard`(*chars*, *typeConf*) $\rightarrow$ `OK`	sets up the keymap characters *chars* and the typing behaviour *typeConf*
`get_trusted_input`(*X*) $\rightarrow \{Sig_t, oi, amount_t\}$	given *X*, where *X* is the raw structure (Table 8.1) for each previous output, returns the signature of each previous output Sig_t, the output index *oi* and $amount_t$ of the previous transaction *t*
`untrusted_hash_transaction_input_start`(Sig_t,*oi*, *amount*) $\rightarrow$ `OK`	streams the inputs, Sig_t, *oi* and $amount_t$ to the dongle using the raw structure (Table 8.1)
`untrusted_hash_transaction_input_finalize`($addr_p$, $amount_p$, $fees_p$, $bipDer_c$, $findex_c$, $lindex_c$) $\rightarrow \{c, addr_p, amount_p, fees_p, pk_c, secFC\}$	streams the outputs, payment address $addr_p$, payment amount $amount_p$, $fees_p$, and selects the key pk_c to which the change will be sent based according to its BIP32 parameters $bipDer_c$,$findex_c$,$lindex_c$. The command returns the change *c*, the change key pk_c, dongle's confirmation of add_p, $amount_p$, $fees_p$, and the characters of the address *secFC* to be authenticated by the user
`untrusted_hash_sign` ($bipDer_i$, $findex_i$, $lindex_i$, *secFR*) $\rightarrow Sig_p$	returns the signature Sig_p of the transaction *p* with key sk_i given its BIP32 parameters $bipDer_i$, $findex_i$, $lindex_i$, iff *secFR* is correct

Table 9.1: The LEDGER commands and their meaning.

tion, the user provides that passphrase when generating that key and each time he wishes to proceed to a transaction. LEDGER wallets use pk_{kp} as a passphrase to generate that encryption key.
(v) $\{sk_{auth}, pk_{auth}\}$: signature/verification keypair for the dongle-API authentication.

LEDGER dongles do not follow the common smart-card file structure: according to ISO 7816 smart-cards support dedicated and elementary files in which they can store private related data *e.g.*, keys. Instead of supporting files and storing the keys, the LEDGER dongles generate on the fly private keys whenever they are requested for signatures, following a tree-like structure in which the master key-pair is the root. The keys are referenced according to their corresponding BIP32 derivation parameters: (1) the number of derivations *bipDer*, (2) the first derivation index *findex* and (3) the last derivation index, *lindex*.

LEDGER operation modes. The LEDGER wallets support a set of different operation modes which describe the allowed functionality of the dongle with regards to the selected security environment that the dongle operates in. The supported modes are:

(*i*) *Standard mode*, which allows standard Bitcoin scripts (addresses staring with 1) or P2PSH scripts (addresses staring with 3) and a single change address. At the beginning of the transaction the user is shown the amount to pay, the change, and any fees. This is the *default* mode of the LEDGER wallets.

(*ii*) *Relaxed mode*, which allows arbitrary outputs to be authorised. At the beginning of a transaction the user is shown the amount to pay.

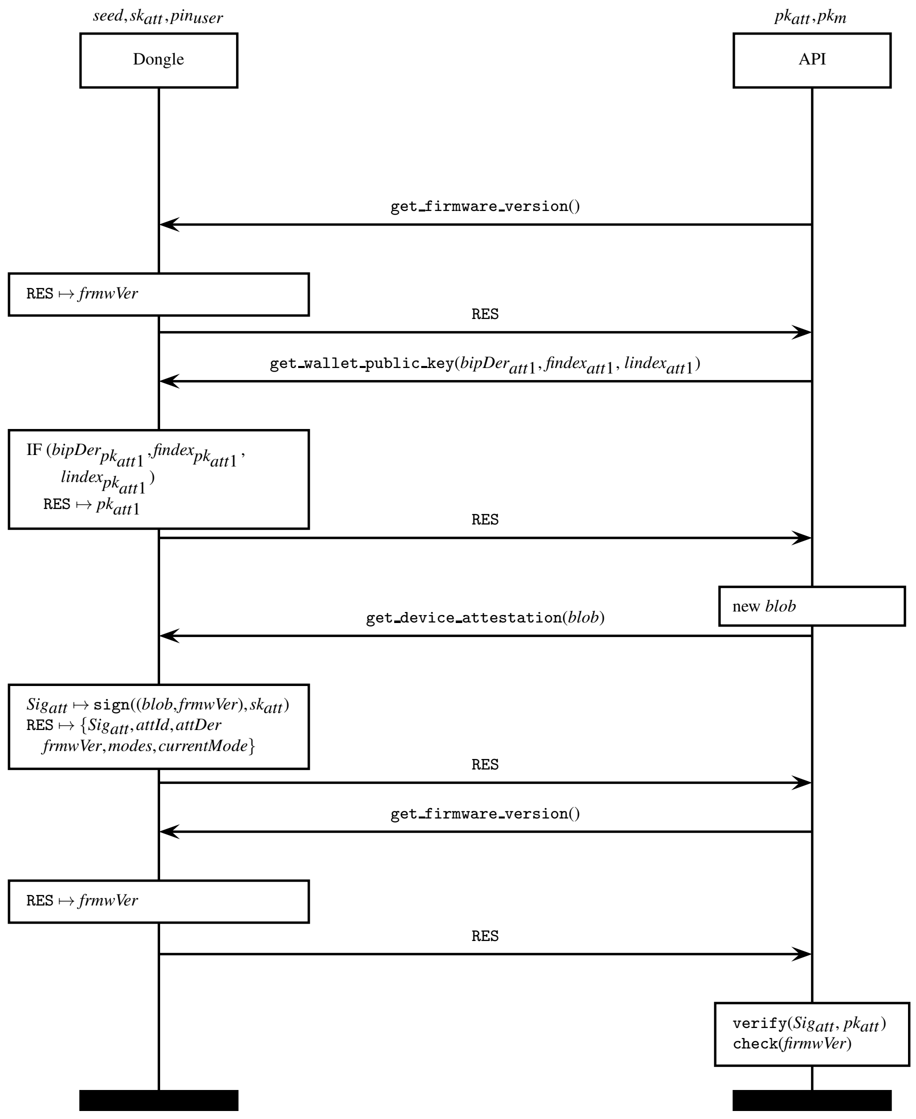
(*iii*) *Server mode*, which allows arbitrary outputs to be authorised but the transactions are controlled by a number of parameters, *e.g.*, maximum total of transactions.

(*iv*) *Developer mode*, which allows arbitrary data to be signed. This mode is used for testing and if enabled then the wallet cannot process regular transactions. The keys that are used when operating at a developer mode are specific ones and they cannot be used for transaction purposes .

Second-factor authentication. Both Ledger Nano and Nano S incorporate a second factor authentication mechanism to ensure that transactions are not tampered. Before signing the transaction, the wallet requests the user's confirmation of the payment address. In Ledger Nano the second factor authentication is of the form of a challenge-response, based on a 58-character-pairs security card the user is provided with. Each time the dongle is requested to process a payment, it presents the user with a challenge *secFC* consisting of four indexes of the payment address. The user responds to that challenge with the corresponding characters from the security card, *secFR*. Only if *secFR* is correct will the dongle continue processing the transaction. Nano S also requires user interaction to process a transaction: before signing the transaction it displays part of the payment address, the payment amount and the fees on its screen. Only if the user confirms the transaction data by pressing the OK button will the dongle sign the signature.

9.1 Dongle Alive Protocol

The protocol consists of four message requests with which the API checks the integrity of the dongle's firmware through an attestation check: the API requests the dongle to sign a random *blob* concatenated with the firmware version *frmwVer* under a manu-



$seed, sk_{att}, pin_{user}$
Dongle
pk_{att}, pk_m
API
get_firmware_version()
RES $\mapsto$ *frmwVer*
RES
get_wallet_public_key($bipDer_{att1}, findex_{att1}, lindex_{att1}$)
IF ($bipDer_{pk_{att1}}, findex_{pk_{att1}}, lindex_{pk_{att1}}$) RES $\mapsto pk_{att1}$
RES
new *blob*
get_device_attestation(*blob*)
$Sig_{att} \mapsto$ sign((*blob*,*frmwVer*),sk_{att}) RES $\mapsto \{Sig_{att}, attId, attDer, frmwVer, modes, currentMode\}$
RES
get_firmware_version()
RES $\mapsto$ *frmwVer*
RES
verify(Sig_{att}, pk_{att}) check(*firmwVer*)

Figure 9.1: The Nano *Dongle Alive* protocol.

facturer key sk_{att}. The exact steps are: (*a*) The API retrieves the dongle's firmware version *frmwVer*. (*b*) The API retrieves pk_{att}. (*c*) The API sends *blob* to the dongle and retrieves the signature Sig_{att} of the *blob* concatenated to the firmware version *frmwVer*, the the attestation key parameters *attId* and *attDer*, *frmwVer* and the operation *modes* and *currentMode*. (*d*) The API retrieves again *frmwVer* and verifies Sig_{att}. The state transition diagram of the protocol can be found in Figure 9.1.

Ledger Nano S. The API retrieves pk_{att}, the dongle's firmware version *frmwVer* in plaintext, sets the currency and retrieves the keys pk_{auth}, pk_{kp}. The Nano S protocol does not include the attestation authentication.

9.2 Login Protocol

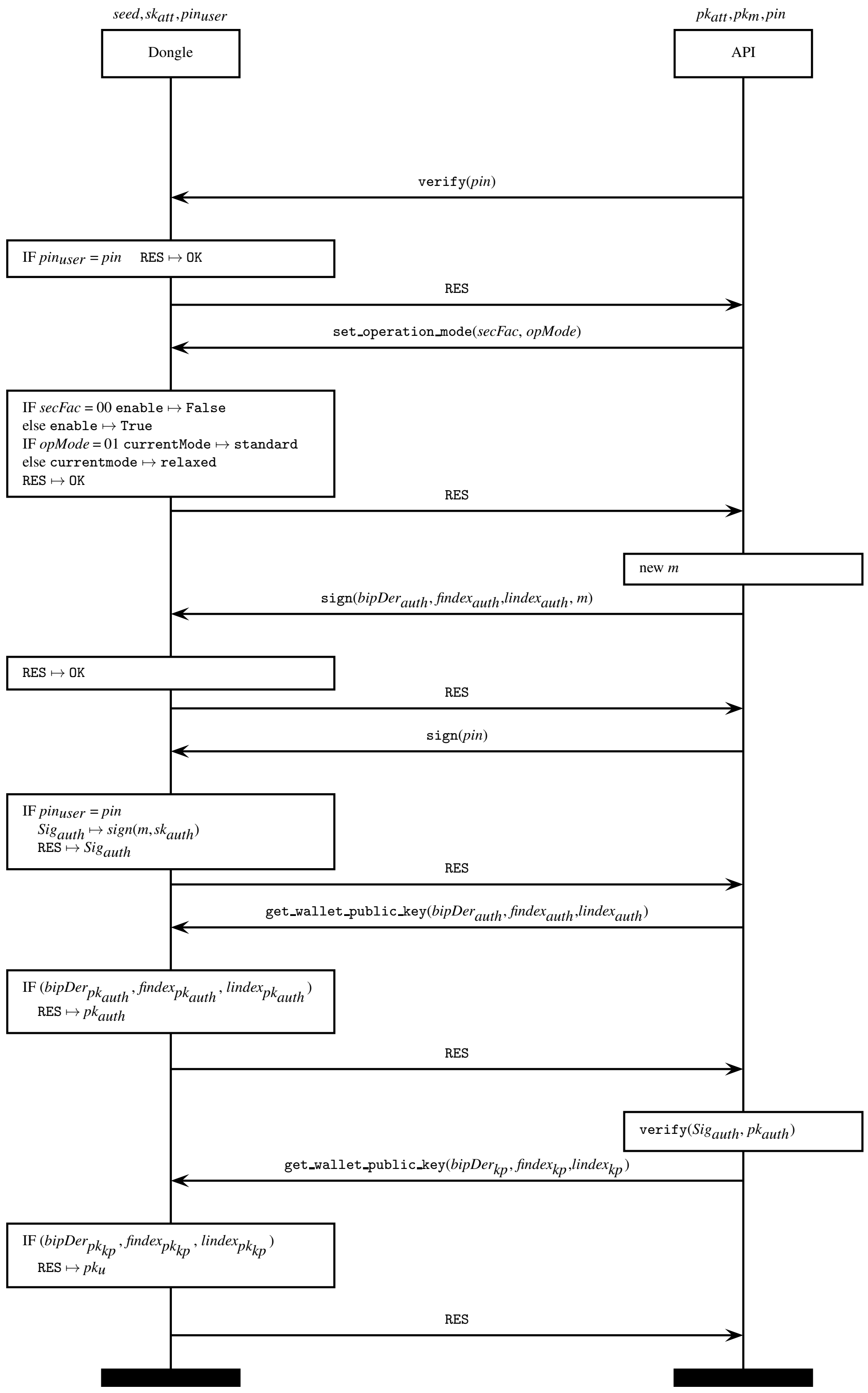

seed, sk_att, pin_user
Dongle
pk_att, pk_m, pin
API
verify(pin)
IF pin_user = pin RES ↦ OK
RES
set_operation_mode(secFac, opMode)
IF secFac = 00 enable ↦ False
else enable ↦ True
IF opMode = 01 currentMode ↦ standard
else currentmode ↦ relaxed
RES ↦ OK
RES
new m
sign(bipDer_auth, findex_auth, lindex_auth, m)
RES ↦ OK
RES
sign(pin)
IF pin_user = pin
Sig_auth ↦ sign(m, sk_auth)
RES ↦ Sig_auth
RES
get_wallet_public_key(bipDer_auth, findex_auth, lindex_auth)
IF (bipDer_pk_auth, findex_pk_auth, lindex_pk_auth)
RES ↦ pk_auth
RES
verify(Sig_auth, pk_auth)
get_wallet_public_key(bipDer_kp, findex_kp, lindex_kp)
IF (bipDer_pk_kp, findex_pk_kp, lindex_pk_kp)
RES ↦ pk_u
RES

Figure 9.2: The Nano *Login* protocol.

Ledger Nano. The *Login* Protocol (Figure 9.2) establishes an authenticated session by which the user gains access to the dongle and, consequently, to the wallet. In contrast to Nano S in which no communication is involved (the user authenticates directly from the device's surface), the protocol consists of six messages, with the main operations being: (*a*) user pin verification, (*b*) dongle authenticity verification via a signature check, and (*c*) retrieval of wallet-related keys. The API also enables or disables a second-factor authentication for payments and configures the wallet's operation modes.

The steps of the protocol are: (*a*) The API sends the user's *pin* to the dongle. (*b*) Upon *pin* verification the API sets the second factor authentication (*SecFac*) and wallet operation (*opMode*) modes. (*c*) The API requests the dongle to sign a random message m with key sk_{auth} and retrieves Sig_{auth} by sending *pin*. (*d*) The API retrieves pk_{auth} and verifies Sig_{auth}. (*e*) The API retrieves pk_{kp}.

Ledger Nano S. The user gains access to the dongle by verifying her PIN through the dongle's interface. There is no need for the dongle to be connected to an active API, nor for the user to re-authenticate when the dongle is plugged in.

9.3 Wallet Setup Protocol

Ledger Nano. The setup process begins API-side. After selecting a PIN, the user is given a 24-word passphrase which corresponds to the wallet's seed. After the user has confirmed the correct passphrase by providing the words that the API has requested, API-side initialisation is done. Then, the dongle-side setup begins. The main operations of the *Setup* protocol (to avoid cluttering Figure 9.3 presents only the exchanged messages that are exploited by our attacks), are: user pin and seed initialisation, and the keyboard and operations mode setup. During initialisation, the API also retrieves the master public key pk_m, and the first derived public key pk_1. The message flow is the following: (*a*) The API sets up a new *pin* and *seed* and requests the generation of $\{sk_m, pk_m\}$. (*b*) The API requests from the dongle to sign *frmwVer* concatenated to a random *blob* using the key sk_{att}. (*c*) The API verifies the *pin*. (*d*) The API retrieves pk_1. (*e*) The *Dongle Alive Protocol* takes place. (*f*) The *Login Protocol* takes place. (*g*) The API retrieves pk_m and some extra unidentified key k_u.

Ledger Nano S. Nano S is initialised offline: the user selects her own PIN and obtains the 24-word mnemonic by communicating directly with the dongle through its interface. After initialisation, when the dongle connects for the first time to an active

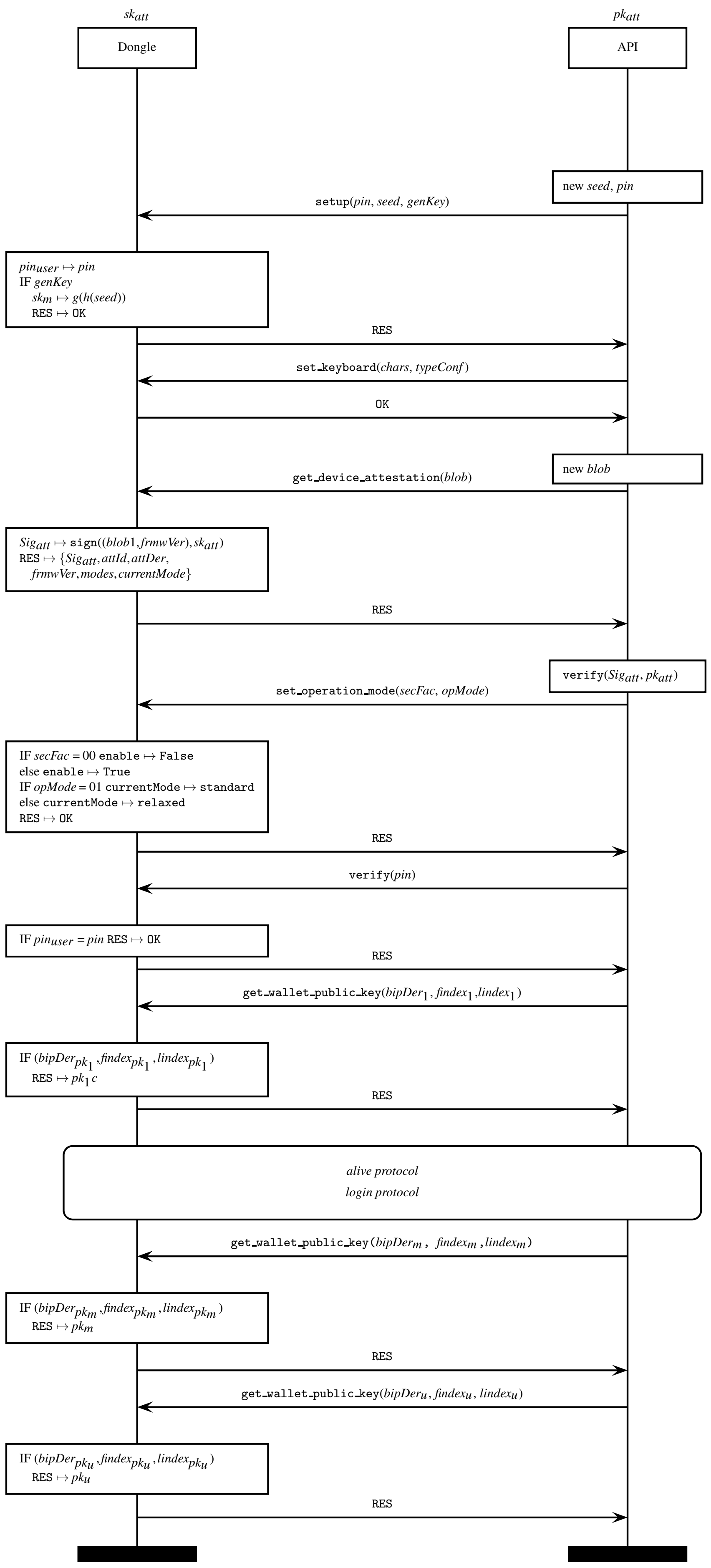
API a *Dongle Alive* session takes place, followed by the authentication of the dongle via its signature. Finally, the API retrieves pk_m.

9.4 Payment Protocol

LEDGER implements a proprietary *Segregated Witness* by enforcing the API to send a detailed description of the inputs before the payment processing: the API forms a pseudo transaction block which has only the inputs, and sends it to the dongle, through a set of `trusted_input` commands. The dongle parses the block (bytewise concatenation) and returns its signature Sig_i. When the API creates the actual transaction, it will use Sig_i to define the corresponding input.

Ledger Nano. The protocol, shown in Figure 9.4, is as follows: (*a*) The API sends to the dongle the available funds through sets of `get_trusted_input` commands. The inputs are sent in the form of pseudo transactions (following the specification in Table 8.1): one for each input. The number of `get_trusted_input` command sets is equal to the addresses $(t_i, i \geq 1)$ with available funds. When the dongle has successfully received block t for a given input, it signs it and returns the signature Sig_t, the output index and the amount. (*b*) The API retrieves pk_t for input t. (*c*) The API creates the actual transaction block (Table 8.1), requested by the user, by sending the inputs Sig_t through sets of `untrusted_hash_transaction_input_start` commands, each set corresponding to a single input. Then, outputs, *i.e.*, the payment address $addr_p$, the payment amount $amount_p$, the fees $fees_p$ and the change key pk_c parameters ($bipDer_c$, $findex_c$, $lindex_c$), are sent via a `untrusted_hash_transaction_input_finalize` command. (*d*) The dongle calculates the remaining balance c, selects the authentication bytes $secFC$ sends back to the API a confirmation of the payment details, c, pk_c and $secFC$. (*e*) The API requests from the dongle to sign the transaction with sk_t by sending the user's validation code, $secFR$. (*f*) The dongle checks $secFR$ against $secFC$ and $addr_p$ and, if it is correct, it computes and returns the transaction signature Sig_t.

Nano S. The *Payment* proceeds as presented in Figure 9.4 with a few differences: (*a*) The API starts the transaction by retrieving the balance address, pk_c, with a `get_wallet_public_key` command. (*b*) The API sends pk_c back to the dongle with the
`untrusted_hash_transaction_input_finalize` command. (*c*) There is no second factor authentication asked by the dongle, or sent by the API.



sk_{att}
Dongle
pk_{att}
API
new *seed*, *pin*
setup(*pin*, *seed*, *genKey*)
$pin_{user} \mapsto pin$
IF *genKey*
$sk_m \mapsto g(h(seed))$
RES ↦ OK
RES
set_keyboard(*chars*, *typeConf*)
OK
new *blob*
get_device_attestation(*blob*)
$Sig_{att} \mapsto$ sign$((blob1, frmwVer), sk_{att})$
RES ↦ $\{Sig_{att}, attId, attDer, frmwVer, modes, currentMode\}$
RES
verify(Sig_{att}, pk_{att})
set_operation_mode(*secFac*, *opMode*)
IF $secFac = 00$ enable ↦ False
else enable ↦ True
IF $opMode = 01$ currentMode ↦ standard
else currentMode ↦ relaxed
RES ↦ OK
RES
verify(*pin*)
IF $pin_{user} = pin$ RES ↦ OK
RES
get_wallet_public_key$(bipDer_1, findex_1, lindex_1)$
IF $(bipDer_{pk_1}, findex_{pk_1}, lindex_{pk_1})$
RES ↦ $pk_1 c$
RES
alive protocol
login protocol
get_wallet_public_key$(bipDer_m, findex_m, lindex_m)$
IF $(bipDer_{pk_m}, findex_{pk_m}, lindex_{pk_m})$
RES ↦ pk_m
RES
get_wallet_public_key$(bipDer_u, findex_u, lindex_u)$
IF $(bipDer_{pk_u}, findex_{pk_u}, lindex_{pk_u})$
RES ↦ pk_u
RES

Figure 9.3: The Nano *Setup* protocol.

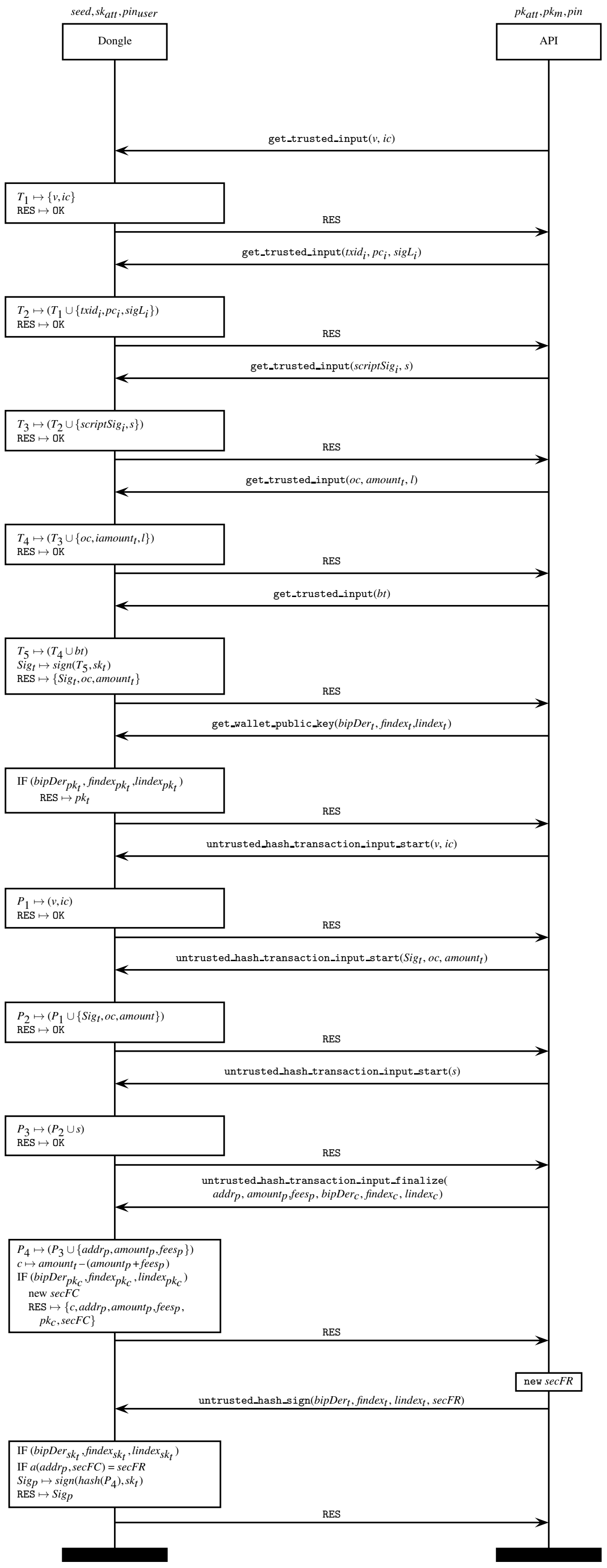

seed, sk_att, pin_user
Dongle
pk_att, pk_m, pin
API
get_trusted_input(v, ic)
T_1 ↦ {v, ic}
RES ↦ OK
RES
get_trusted_input(txid_i, pc_i, sigL_i)
T_2 ↦ (T_1 ∪ {txid_i, pc_i, sigL_i})
RES ↦ OK
RES
get_trusted_input(scriptSig_i, s)
T_3 ↦ (T_2 ∪ {scriptSig_i, s})
RES ↦ OK
RES
get_trusted_input(oc, amount_t, l)
T_4 ↦ (T_3 ∪ {oc, iamount_t, l})
RES ↦ OK
RES
get_trusted_input(bt)
T_5 ↦ (T_4 ∪ bt)
Sig_t ↦ sign(T_5, sk_t)
RES ↦ {Sig_t, oc, amount_t}
RES
get_wallet_public_key(bipDer_t, findex_t, lindex_t)
IF (bipDer_pk_t, findex_pk_t, lindex_pk_t)
RES ↦ pk_t
RES
untrusted_hash_transaction_input_start(v, ic)
P_1 ↦ (v, ic)
RES ↦ OK
RES
untrusted_hash_transaction_input_start(Sig_t, oc, amount_t)
P_2 ↦ (P_1 ∪ {Sig_t, oc, amount})
RES ↦ OK
RES
untrusted_hash_transaction_input_start(s)
P_3 ↦ (P_2 ∪ s)
RES ↦ OK
RES
untrusted_hash_transaction_input_finalize(
addr_p, amount_p, fees_p, bipDer_c, findex_c, lindex_c)
P_4 ↦ (P_3 ∪ {addr_p, amount_p, fees_p})
c ↦ amount_t − (amount_p + fees_p)
IF (bipDer_pk_c, findex_pk_c, lindex_pk_c)
new secFC
RES ↦ {c, addr_p, amount_p, fees_p, pk_c, secFC}
RES
new secFR
untrusted_hash_sign(bipDer_t, findex_t, lindex_t, secFR)
IF (bipDer_sk_t, findex_sk_t, lindex_sk_t)
IF a(addr_p, secFC) = secFR
Sig_p ↦ sign(hash(P_4), sk_t)
RES ↦ Sig_p
RES

Figure 9.4: The Nano *Payment* protocol.

Chapter 10

Attacks on the Ledger Wallets

10.1 General Threat Model for the Bitcoin Wallets

A Bitcoin wallet should provide high levels of security and privacy for the user, while also being easy to use. We therefore consider a wallet to be secure when it provides: (*a*) guarantees against tampering, (*b*) a secure environment for transaction processing, and (*c*) account privacy.

Our threat model assumes perfect cryptography *i.e.*, we do not look into possible vulnerabilities at the underline cryptographic algorithms. We consider the wallet's code running on the smart card as well as the code running in the API to be trusted. We assume these pieces of code not to be malicious, and trust them not to steal sensitive data such as pins or keys. The adversary has full control of the communication layer between the dongle and the API (he can eavesdrop and tamper the communication by deleting, inserting and altering the messages) and targets the user's account having one or both of the following goals:

(i) Access to the the user's funds. This entails getting the ownership of the account (*i.e.*, possession of the master private key), impersonating the user (*i.e.*, possession of the user's PIN), by changing the wallet's security parameters, and/ or tampering the transactions while the wallet process them.

(ii) Track the account's transactions. This entails access to the public keys of the wallet, either by possessing the master public key and/or by eavesdropping the processed transactions. Based on the adversary's goals, we categorise the possible threats of Bitcoin wallets to *Direct wallet*, *Transaction* and *Account privacy*, to define different attack scenarios. Table 10.1 presents the attacks that belong to each category. As we show now, the LEDGER wallets are subject to *all* such attacks.

a. Direct wallet attacks	***b. Transaction attacks***	***c. Account privacy attacks***
a.1 access to the master private key sk_m;	b.1 tamper the payment amount;	c.1 account traceability.
a.2 access to the key pool encryption key;	b.2 tamper the payment address;	
a.3 unauthorised access to the wallet;	b.3 denial of service.	
a.4 alter the wallet security properties.		

Table 10.1: Attack categories.

10.2 Summary of Attacks

This Section presents the instantiation of the attacks discussed in section 10.1, on LEDGER wallets. We show how we were able to perform attacks from the APDU layer, by bypassing the restrictions of the API. Some attacks are passive, *i.e.*, they only require observing the communication channel; while others are active *i.e.*, they involve relaying and altering the exchanged messages.

10.2.1 Direct Wallet Attacks

LEDGER dongles aim to provide a fully isolated and secure environment for all cryptographic and Bitcoin transaction operations. They specifically incorporate the smart-card technology, as they intend to manage and store keys internally. Our experiments show how these wallets can be compromised without tampering the hardware.

a.1: Accessing the account's master private key sk_m. Access to the wallet's seed s is synonymous to having access to sk_m. During the *Setup* protocol execution we were able to sniff s which was sent in plaintext from the API to the dongle. By using the BIP32 derivation function we regenerated sk_m and all children keys. The API having access to s and transmission of s in plaintext defeats the purpose of cold storage. The attacker may gain access to the *Setup* protocol, and consequently to s, by forcing the dongle's reinitialisation.

Mounting the Attack a.1 Given the user's pin p, a replay of the session $\{$`verify`$(p') \rightarrow$ *ERROR*, `verify`$(p') \rightarrow$ *ERROR*, `verify`$(p') \rightarrow$ *ERROR*$\}$ in which an incorrect pin p' is sent to the dongle three consecutive times, has as a result the dongle to enter a lock state. In the lock state the dongle is inaccessible and produces a request for initialisation.

a.1.1: Importing the attacker's master private key sk_a. Knowledge of the *Setup* protocol allows a key-injection attack; the adversary access a legitimate *Setup* session by mounting the Attack a.1 and then imports his own key by mounting the Attack a.1.1. After that, the user will use the attacker's keys and the corresponding addresses instead of his own. Such an attack is more likely when the receiving money is usually greater than the spending (*e.g.*, stores).

Mounting the Attack a.1.1 Given a session of the *Setup* protocol $\{$`setup`$(p,s) \rightarrow$ *OK*, `set_keyboard`$(c,\ tc) \rightarrow$ *OK* ... `get_wallet_public_key`$(pk\_params) \rightarrow pk\}$ as presented in Figure 9.3, the attacker locks the dongle with Attack a.1 and replays that session by applying the substitution $(\mapsto)$: `setup`$(p,\ s \mapsto s_a)$ where s_a is the attacker's seed. At the end of the session, the dongle will generate the master keypair $\{sk_a, pk_a\}_m$ according to s_a, and consequently, all keys after the *Setup* session will be derived from the attacker's seed.

a.2: Accessing the key-pool. Unauthorized access to the key-pool implies loss of privacy and account tracability as the adversary gains insight on the addresses that the account uses/has used. During the *Login* protocol, we sniffed the passprase that is used to create the key-pool key. This passphrase is the public key pk_{kp} and is retrieved in plaintex after a `get_wallet_public_key` command.

a.3: Unauthorised access to the wallet. A general requirement in Bitcoin wallets is to be used only by users that have the credentials, *e.g.*, the pin. The *Login* protocol that we extracted indicates that each time the user connects to the dongle (though only in the LEDGER Nano case) the *pin* is sent in plaintext to the dongle via a `verify` command. As such the wallet is vulnerable to eavesdropping attacks of the user's credentials and consequently unauthorised access to it and to the account.

a.4.1: Tampering the wallet security properties. Each transaction is secured by a second factor authentication, where the user has to verify specific characters of the payment address. The verification is done through challenge-response: the dongle requests the characters to be verified by providing their indexes within the address *secFC*, and the user provides the corresponding characters of the security card *secFR*. The `set_operation` command enables/disables the second factor authentication mechanism for future transactions. Although this command appears in every *Login* session, it is only during the *Setup* protocol that it has a direct effect on the dongle. An attacker can access the *Setup* protocol by mounting the Attack a.1 which locks the dongle and forces reinitialisation.

Mounting the Attack a.4.1 Given a legitimate session of the *Setup* protocol, {`setup(`p`,`s`)`$\rightarrow$ *OK*, ..., `set_operation_mode(`*enable*, *standard*`)`$\rightarrow$ *OK*, ...} as presented in Figure 9.3, replay that session by applying the substitution ($\mapsto$): `set_operation_mode(`*enable* $\mapsto$ *disable*, *standard* $\mapsto$ *relaxed*`)`. After that session, the dongle is not requesting for the user's validation of the payment address. In each session of the *Login* protocol (as presented in Figure 9.2), relay the communication and perform the following substitution `set_operation(`*enable* $\mapsto$ *disable*, *standard* $\mapsto$ *relaxed*`)`. In each session of the *Payment* session (as presented in Figure 9.4), relay the communication by applying the following substitutions ($\mapsto$):
i) `untrusted_transaction_input_hash_finalize`:
`response(`c,$addr_p$,$amount_p$, pk_c, $no \mapsto secFC$`)` where *no* is the card's response that no second authentication is required, and *secFC* are four random characters of the payment address $addr_p$.
ii) `untrusted_has_sign(`*sk_params*, *secFR* $\mapsto$ *no*`)` where *secFR* is the user's input according to *secFC* and *no* defines that no secondary authentication took place.

That attack changes the security parameters of the dongle so that during a transaction processing the user's validation of the address is no longer required. To successfully apply a payment address attack there is some further command tampering (explained in detail in Section 10.3) that needs to be mounted.

a.4.2: Learning the security card. Another way to successfully attack the payment destinations is to gain access to the security card that is used by the user to validate the payment address. The security card consists of 58 hexadecimal characters that encode the letters A-W, a-w and the numbers 0-9. Eavesdropping a *Payment* session reveals at least four security card mappings. Our experiments indicated that the dongle always request at least a new index in each *Payment* session until all keycard characters have been requested at least once. So, in the worst-case scenario, an adversary may learn a single new character in each round of the *Payment* protocol, so, after 58 legitimate rounds he knows all characters of the security card. In the best-case scenario, each time four new characters will be asked so the attacker will need 15 legitimate *Payment* sessions without relaying the communication at all. This can be enforced by tampering the communication so that the user provides input for mappings that are still unknown.

i) the adversary alters $secFC \mapsto secFC'$ in favour of the character mappings he does not know, ii) the adversary returns to the dongle the correct *secFR* according to the original challenge *secFC*.

Mounting the Attack a.4.2 Given a legitimate session of the *Payment* protocol {`get_trusted_input`$(v, ic) \rightarrow OK, \ldots,$ `untrusted_hash_transaction_finalize`$(addr_p, \ldots) \rightarrow (c, \ldots, secFC)$, `untrusted_hash_sign`$(sk_p arams, secFR) \rightarrow OK$} an attacker knows the characters of the payment address, $padd_p$, the characters of the address to be validated, *secFC*, the user's input based on the keycard, *secFR*, and whether the *secFR* is correct (the dongle responds with *OK* to the `untrusted_hash_sign` command). If the pair *secFC*/*secFR* is already known from a previous session, the attacker can learn a new pair $secFC'/secFR'$ by relaying the communication and applying the following substitutions ($\mapsto$): { `get_trusted_input`$(v, ic) \rightarrow OK, \ldots,$ `untrusted_hash_transaction_finalize`$(addr_p, \ldots) \rightarrow (c, \ldots, secFC \mapsto secFC')$, `untrusted_hash_sign`$(sk_p arams, secFR' \mapsto secFR) \rightarrow OK$}.

10.2.2 Transaction Attacks

b.1-b.2: Tampering the Bitcoin transactions. The processing of the payment data is achieved through the `untrusted_hash_transaction_input_finalize` command-response pair, as shown in Table 10.2. The API sends to the dongle the payment address $addr_p$, the payment amount $amount_p$, the fees of the transaction $fees_p$ and the BIP 32 parameters of the public key from which the change address will be derived pk_c *parameters*. When the dongle receives the command, it processes the data and returns to the API the remaining balance[1] of the account (change) c, a confirmation of $addr_p$ and $amount_p$, the public key of the change pk_c, and the indexes of the address's characters *secFC* that the user needs to validate.

Given a *Payment* session an adversary can (a) redirect the payment destination: $addr_p$ and (b) tamper the payment amount: $amount_p$ by altering the exchanged messages. For an adversary to successfully mount a payment redirection, he needs to have access to the mappings on the security card. In that case, the communication tampering is bidirectional: i) when the API sends the payment data to the dongle the adversary

[1] $c \mapsto (balance - (payment_p + fees_p)$.

APDU	traces
command	untrusted_transaction_input_hash_finalize($addr_p$, $amount_p$,$fees_p$, pk_c *parameters*)
response	(c, $addr_p$, $amount_p$, pk_c, $secFC$)

Table 10.2: Abstraction of the `untrusted_hash_transaction_input_finalize` command-response pair.

substitutes the payment address with the one he desires, ii) when the dongle responds with the payment data, the adversary changes the payment address to the original one. The consequence is the API requesting from the user the validation of four characters of the original payment address. When the API requests the signature of the payment it sends together the four mappings of the characters that the user verified. The adversary then relays again the communication and substitutes these four mappings with the ones that correspond to the tampered address.

Mounting the Attack b.1-b.2 Depending on the payment data of interest, by relaying a *Payment* session, the following attacks can be mounted:
i) to redirect the payment destination apply the following substitutions ($\mapsto$):
`untrusted_transaction_input_hash_finalize(` $addr_p \mapsto addr'_p$, $amount_p$,$fees_p$, pk_c *parameters*) $\rightarrow$ `response(`c, $addr_a \mapsto addr_p$, $amount_p$, pk_c, $secFC$). In the command data the original payment address $addr_p$ is substituted by the attacker's address add'_p. The response is also relayed so that it contains the original address. `untrusted_hash_sign(`$bipDer_t$,$findex_t$, $lindex_t$, $secFR \mapsto secFR'$) The adversary substitutes the user's input which is according to $payment_p$ with the corresponding mappings of $payment'_p$

ii) to tamper the payment amount apply the following substitutions ($\mapsto$):
`untrusted_transaction_input_hash_finalize(` $addr_p$, $amount_p \mapsto amount'_p$, $fees_p$, pk_c *parameters*) $\rightarrow$ `response(`$c' \mapsto c$, $addr_a \mapsto addr_p$, $amount'_p \mapsto amount_p$, pk_c, $secFC$). In the command data the original payment amount $amount_p$ is substituted by the attacker's amount $amount'_p$ whereas in the response data $amount'_p$ is changed back to $amount_p$ and the remaining funds c' is changed to the amount that would result after the original payment amount.

The attack can also be mounted without tampering the `untrusted_hash_sign` command if i) the second factor authentication is disabled (Attack a.4), or ii) by letting the user know the tampered address and expect the user to validate it due to human error factors.

b.3: Denial of service. DoS attacks that target specific Bitcoin Wallet users have become viral *e.g.*, the DoS attacks on the BitGo wallets, leaving many users unable to use their funds. Such attacks target the wallet's server and usually consist of sending a huge amount of requests. Although this is out of the scope of our practical experiments, in the LEDGER wallet side of things, DoS attacks could also be mounted from the APDU layer by tampering the transaction data in a way that either the dongle cannot interpret it, or that the transaction cannot be verified in the network (*e.g.*, by tampering the verification key).

10.2.3 Privacy Attacks

c.1 Account traceability. Although Bitcoin's core intent is not privacy, it is associated with anonymity and is often used by users who want their actions to be unlinkable. HD wallets like LEDGER allow the creation of a sequence $pk_1, pk_2, .., pk_n$ of child public keys directly from the master public key pk_m with the formula $pk_i = f(hash(i, pk_m))$ where i is the child key index and f the generator function. As such, all receiving and balance addresses are derived from pk_m. Leakage of pk_m entails loss of privacy, as access to pk_m is equal to having access to all addresses generated by the wallet.

Mounting the Attack c.1 During a legitimate session of the *Setup* protocol, {`setup(`p`,`s`)`$\rightarrow OK$, `set_keyboard(`c`, `tC`)`$\rightarrow OK \ldots$, `get_wallet_public_key(`pk_m_params`)`$\rightarrow pk_m$} as presented in Figure 9.3, pk_m is transmitted in plaintext.

10.3 Technical Details of the Attacks

This Section provides a detailed description of the active APDU middle attacks. Passive attacks are not described in detail as the only prerequisite is to sniff a legitimate session and analyse the traces according to Chapter 9. All the attacks, apart from those that require replaying a *Setup* session, are automated and can be applied through the man-in-the-middle tool we developed.

Disabling the second factor authentication for LEDGER Nano. A second factor authentication based on the user's input is utilised when the dongle processes a transaction. LEDGER uses this mechanism in the Nano dongles as a measure against payment address tampering. The assumption is that by asking the user to validate those characters, he will be able to identify possible alterations to the address. The

2nd authentication	**command bytes**
✓	`e04800001f058000002c800000008000000000000000000000000001` **040c060006** `0000000001`
×	`e04800001f058000002c800000008000000000000000000000000001` **00** `0000000001`

Table 10.3: The `untrusted_hash_sign` command. The bytes in bold indicate the authentication parameters.

goal of the attack explained in this section is to successfully disable that authentication mechanism so that the user will not be able to identify the payment address tampering.

After receiving the available funds, the dongle receives the payment details (address, amount *etc.*) through an `untrusted_hash_transaction_input_finalize`. Then, the dongle chooses four characters from the address that require validation, and includes their indexes to its response: the last five bytes of the dongle's response to an `untrusted_hash_transaction_input_finalize` are of the form `02 xx xx xx xx` where `02` denotes that a second factor authentication is required and `xx xx xx xx` denote the indexes of the chosen characters within the address, *secFC*. Table 10.5 presents an example of the structure of that response with respect to *secFC*. The API then requests the user's input *secFR*, according to *secFC* and provides it to the dongle together with the signature request via an `untrusted_hash_sign` command. Table 10.3 presents the difference in an `untrusted_hash_sign` command when the second factor is enabled and disabled.

The command that enables/disables the authentication parameters of the LEDGER dongles is the `set_operation` command. Although during the *Login* protocol the API sends this command, our experiments showed that this command effectively changes the dongle's authentication parameters only when it is sent during the *Setup* protocol. Also, for the *disable-the-second-factor-authentication* command to be successful the dongle's operation mode must be switched from *standard* to *relaxed*[2] . Table 10.4 shows a comparison between the original `set_operation` and the one used for the attack. In the original command the P1 byte is `01` which specifies that the authentication is enabled, P2 is `01` is a dummy code *i.e.*, changing it to any value does not affect the command, and Data is `01` which specifies that the standard wallet mode is selected. In the command we used we altered these bytes so that P1 is equal to `00` which specifies that the authentication is disabled, P2 equal to `00` as defined by the LEDGER standard

[2]The *standard* mode always requires the user's validation of the payment address; The *relaxed* mode does not require the user's validation of the payment address.

command	**Cla**	**Ins**	**P1**	**P2**	**Lc**	**Data**
original	e0	26	01	01	01	01
attack	e0	26	00	00	01	02

Table 10.4: The `set_operation` command.

2nd authentication	**response bytes**

✓	`450203b1000000000000001976a914f1253f0463e5877c5e8bb3`
	`f34e7abfb335023ee188ac0553000000000000001976a914e6e4`
	`4d66125327341d6abb71e0702a4ea053743788ac` **024040f050**

×	`450203b1000000000000001976a914f1253f0463e5877c5e8bb3`
	`f34e7abfb335023ee188ac0553000000000000001976a914e6e4`
	`4d66125327341d6abb71e0702a4ea053743788ac` **00**

Table 10.5: The dongle's response to a `untrusted_hash_transaction_input_finalize` command. The bytes in bold indicate the authentication parameters.

and Data equal to `02` which specifies that the relaxed wallet mode is selected.

We tested this command by sending it while no protocol was executed and also during the execution of the *Login* protocol[3]. In both cases, the dongle accepted the command but the second factor authentication was not successfully disabled. However, the attack was successful when we replayed a session of the *Setup* protocol and injected the altered command. A *Setup* session can be initialised only if the dongle is locked and the user cannot access it. A sample trace of a successful attack, up to the point in which the modes are set, is shown in Table 10.6. In particular the steps that are required to successfully disable the second factor authentication of the dongle are the following: i) block the dongle by sending three consecutive PINs: we sent three `verify` commands with an altered PIN: $p \mapsto p'$ ii) replay a *Setup* session: we replayed the commands that were sent in a legitimate session to the dongle apart from the `set_operation` command which we altered as shown in Table 10.4. After the replay of the *Setup* session the dongle no longer requires a second factor authentication for the transactions.

After disabling the second factor authentication, during a session of the *Payment* protocol the dongle's response to an `untrusted_hash_transaction_input _finalize` is different; instead of including the five byte-code `02 xx xx xx xx`, it includes the single byte `00` which denotes that no further authentication is needed. However, the API's implementation always considers the user's authentication of the payment address to be obligatory, regardless of the dongle's response. Thus, when the API

[3]The command appears in each session of the *Login* protocol

steps	**APDU traces**
block the dongle	`verify`(p'): 02200000433333333 `verify`(p'): 02200000433333333 `verify`(p'): 02200000433333333
replay a Setup session	`setup`(p,s): e02000004c020a000504313432340040 8c3937fafb22e5f4979e90afe0b912cc05d92b9910c622887f6 1b30d9814f714df2dd5ada8cc5cd663e998dec1cc55915377352 cf6949a20ba444039219efd6900 `set_keyboard`: e028000077000000000000000000000000760f 00d4ffffffc70000007 82c1e3420212224342627252e362d3738271e1f202122232425263 333362e37381f0405060708090a0b0c0d0e0f10111213141516171 8191a1b1c1d2f3130232d350405060708090a0b0c0d0e0f1011 12131415161718191a1b1c1d2f313035 `get_device_attestation`: e0c200000861255ccee7f8c72d `set_operation`: **e02600000102** ...

Table 10.6: Attack traces: disabling the second factor authentication during a *Setup* session.

receives the `00` code the transaction freezes. Another complication of the described attack is that each time a *Login* session takes place, the API sends a `set_operation` command that activates the second factor authentication, leading to an error code response by the dongle. Thus, the attack is further complemented by the following steps:

- in each session of the *Login* protocol, alter the `set_operation` command as presented in Table 10.4,

- alter the part of the dongle's response to the `untrusted_hash_transaction_input_finalize` command which specifies the second factor authentication: `00` $\mapsto$ `02 xx xx xx xx` where `xx xx xx xx` are four random indexes of the address (as shown in Table 10.5),

- alter the part of the `untrusted_has_sign` command that provides the user's input *secFR*: `04 yy yy yy yy` $\mapsto$ `00` (as shown in Table 10.3).

Attacks on Bitcoin transactions. A transaction is processed by the dongle, as discussed in Chapter 9, via a series of consecutive APDU commands: the API sends the transaction related data (*i.e.*, input funds, payment details) and requests the dongle to sign the payment using the corresponding *sk* of the input funds that will be consumed. A payment attack does not require altering all parts of this communication but only the parts that send the payment details to the dongle. Thus, tampering the

APDU	**traces**
command	`e046020048`
	`22314e3371757233596565334b664e74436a4677756e346f`
	`366f4c324478686747796f00000000000530500000000000`
	`001d60058000002c80000000800000000000000100000002`
response	`450203b10000000000001976a914f1253f0463e5877c5e8bb3attack`
	`f34e7abfb335023ee188ac05530000000000001976a914e6e4`
	`4d66125327341d6abb71e0702a4ea053743788ac024040f050`

Table 10.7: `untrusted_hash_transaction_input_finalize` command-response trace.

`untrusted_hash_transaction_input_finalize` command-response data is sufficient. Table 10.7 presents a trace of the `untrusted_hash_transaction_input _finalize` command-response pair. The structure of the command is the following:

`Cla`	`Ins`	`P1`	`P2`	`Lc`	**Data**
`e0`	`46`	`02`	`00`	`48`	$addr_p, amount_p, fees_p,$ pk_c `parameters`

The Data part consists of i) the $addr_p$ bytes which denote the recipient of the payment amount. i) the $amount_p$ bytes which denote the payment amount. i) the $fees_p$ bytes which denote the fees amount for that particular transaction. i) the pk_c `parameters` bytes which denote the BIP32 parameters of the public key whose hash will be the address to which the remaining balance (if any) of the account will be sent.[4]

If the command is successful, the dongle's response has the following structure:

SW1	`SW2`
$amount_p$, $hash160(addr_p)$, c, $hash160(addr_c)$, $secFC$	`OK`

The data part of the command, SW1, consists of: i) the $amount_p$ bytes which are the same bytes as the ones in the command, and serve as a confirmation to the API. i) the $hash160(address_p)$ bytes which denote the hash160 of the payment address $address_p$. i) the c bytes: denote the remaining balance of the account (change). i) the $hash160(addr_c)$ bytes which denote the hash160 of the change address $addr_c$. i) the $secFC$ bytes which denote the indexes of the $address_p$ characters that the user needs to verify. If the second factor authentication mechanism is disabled then these bytes are absent.

[4]The API always requests the generation of a new pk_c even if the remaining balance equals to zero.

10.4 Generality of the Attacks

The purpose of our work is to show that it is possible to attack Bitcoin hardware wallets via the low-level communication. The threat model we present is hardware/software independent and applicable to all available Bitcoin wallets. The attacks on the LEDGER wallets aim to prove that Bitcoin transactions are vulnerable, even if tamper-resistant hardware such as smart-cards are incorporated. Our work showcases how the API restrictions can be bypassed by relaying the hardware communication. The same attacks, adapted to meet the criteria of each hardware, can be potentially applied to every wallet that does not use a secure communication channel *i.e.*, Trezor and Keepkey. All hardware wallets follow the same abstraction of the *Payment* protocol; any plaintext communication is prone to attacks b.1-b.2. Currently all Bitcoin wallets base their entire security on a trusted path between the dongle and the user, by requesting the user to verify the payment data. This trusted path cannot be tampered and the success of our attacks boils down to the user's capability to identify the tampered data. However, previous studies have shown that a significant average of 15% of such verification is usually erroneous. Although most Bitcoin wallets claim to secure the transactions by enforcing the user's validation of the payment data, the success rate of transaction attacks is proportional to the user error rate. The validation/comparison of hashes by the user is a common techniques *e.g.*, device pairing, self-signed certificates with HTTPS *etc.*. The usability aspects of hash comparison in security protocols and the effects of human errors have been studied before. For example, in [Uzun et al., 2007] the authors conclude that the compare-and-confirm method (the user has to confirm a checksum presented on the device's screen) for a 4-digit string has 20% failure rate, whereas the work in [Hsiao et al., 2009] concludes that comparison of the Base32[5] hashes has an average 14% failure rate. Although such studies focus on low entropy hashes, they conclude that, on its own, such a technique cannot provide strong security guarantees given the human error. As such, a transaction attack on an HID wallet depends on whether the user's ability to identify the tampered data.

The privacy issues we address for the LEDGER wallets is an aspect that applies to all BIP32 wallets, especially to those that do not communicate in a secure way. Currently, all hardware wallets[6] transmit the public keys (including the master public

[5]Base32 hashes are a total of 25 bit entropy and consist of five characters with 32 possible character mappings. A Bitcoin address has 160 bit entropy.

[6]Apart from Digital BitBox whose specifications are not available publicly.

data	a.1	a.2	a.3	a.4	b.1	b.2	b.3	c.1
s	✓	✓	✓	✓	✓	✓	✓	✓
pin	×	×	✓	×	×	×	×	×
$secFC, secFR$	×	×	×	✓	×	✓	✓	×
$opMode$	×	×	×	✓	×	✓	✓	×
$addr_p$, $amount_p$,$fees_p$,c,pk_c	×	×	×	×	✓	✓	✓	✓
$pk\_\{m, m+1.., m+n\}$	×	✓	×	×	×	×	×	✓

Table 10.8: Vulnerable data in LEDGER wallets and the corresponding attacks.

key) in the clear: eavesdropping a single session reveals at least two public keys, the address with available funds and the address that the remaining balance will be sent to. Also, whenever the hardware connects to a fresh API, the master public key pk_m is sent in the clear. Access to that key implies access to all children public keys, which makes eavesdropping that single session sufficient to track the account's transactions. In any case, whether the adversary has access to pk_m or to its children pk_i the flow of the funds of the given account is linkable.

10.5 Proposed Lightweight Fixes of the Protocols

The LEDGER wallets, as with all other hardware wallets not using a secure communication channel, fail to prevent MitM attacks. All transaction data is sent in the clear, making the wallet vulnerable to attacks and account linkability. Encrypting the entire communication would be an obvious solution to that. However, such a strategy requires computational power, and possible changes to the security architecture of the current wallets. Additional delays to the transaction processing would be another trade-off. Instead we propose the symmetric encryption of specific communication parts: those that are prone to attacks with respect to our threat model. Table 10.8 summarises what LEDGER wallet data needs to be protected to defend against which attacks. Our fix consists of three components: 1. the secure pre-setup phase, 2. the authentication and session key establishment protocol, 3. encryption of sensitive parts.

Secure environment for the PIN exchange. The PIN needs to be entered in the hardware before the initialisation of the wallet as the PIN is then used to derive the cryptographic keys to protect the interactions between the dongle and the API. This process must occur in a secure offline environment. This can be achieved either by entering the PIN directly on the trusted user interface of the device (if it is an HID wallet), or by setting up the PIN on an air-gapped machine, *e.g.* using a live OS on a USB stick which will ensure that the OS has and will never be connected to the Internet.

Authentication and session key establishment. This protocol gets executed every time the API establishes a new session with the dongle. It is responsible for the API/hardware authentication and the establishment of a fresh session key. A new session is established whenever the hardware connects to an active API. For the key establishment we propose a Password Authenticated Key Exchange by Juggling protocol (j-PAKE) [Hao and Ryan, 2010] which allows bootstrapping high-entropy keys from the low-entropy user's PIN. In that way, we avoid storing secret data API side, ensure that fresh keys are used in each session and guaranteeing the user's presence at that session. In addition, the j-PAKE protocol allows zero knowledge proof of the PIN which satisfies the authentication prerequisites of the session. Finally j-PAKE provides guarantees against off-line and on-line dictionary attacks and it satisfies the forward secrecy and known-key security requirements. J-PAKE, like the Diffie-Hellman key exchange, uses ephemeral values but proceeds in an additional round which combines them with the user's PIN and makes certain randomisation vectors vanish.

Encryption of sensitive data. Once the session key is established, slightly modified versions of the four LEDGER protocols (*Alive*, *Login*, *Setup*, and *Payment*) can be executed. The four new protocols are derived from the original Ledger protocols as follows. First a session identifier is established for each execution of each of these protocols. This will be generated dongle side, and transmitted to the API in plaintext. The session identifier does not need to be confidential, but will need to be fresh and generated by the dongle to avoid replay attacks. Then the dongle and API execute the original protocol but encrypting under the current session key the sensitive data identified previously (Table 10.8). The computed ciphertexts will all include the established session identifier. A Message Authentication Code (MAC) is further computed and concatenated to the chiphertext. The other party will then be able to decrypt and verify the encrypted parts.

10.6 Summary

Although the security of financial related hardware in other areas has always attracted a lot of attention, eg., the Chip and PIN systems [Murdoch et al., 2010b], Bitcoin-related hardware has not been studied before. Relying on the high levels of security that the Bitcoin protocol offers is not enough to guarantee safe transactions. Lack of a standard that defines the properties of the Bitcoin wallets leads to security misconceptions and ad-hoc implementations that hide vulnerabilities. The work presented, to the best of our knowledge, is the first to address security aspects of Bitcoin wallets and stress the importance of securing the implementations of low-level communications. We chose to analyse smart-card based wallets as they are perceived to be the most secure and tamper resilient means for key management. However, the core idea of the attacks is general and applies to other hardware wallets of different technology.

In this work we extract and analyse the protocols that are hidden behind the LEDGER wallets, the only available smart-card based solutions. Our work includes the analysis of both standard and HID dongles. To that end, we identify and categorise possible vulnerabilities for Bitcoin wallets and we introduce a general threat model. We then use that model to analyse the LEDGER protocols. Our experimental work concluded that the LEDGER implementations are vulnerable to a set of attacks that target the wallet itself as well as the Bitcoin transactions. Finally, we propose a lightweight fix, based on the j-PAKE protocol, which can easily be adapted to any wallet and efficiently prevents any active or passive attack. Modelling and evaluating the suggested fix is not within the scope of this thesis; we leave it to future work to address the effectiveness of the suggested fix through formal analysis of the protocol *i.e.*, modelling the communication protocol with the suggested fix and proving the attacks to be impossible.

Attacking the LEDGER wallets is just an example, as the same methodology can be easily adopted in other technologies. The work presented in the Bitcoin Chapters does not aim at proving the specific wallets insecure, but rather to showcase the importance of ensuring a secure low-level implementation even if the higher levels bring guarantees.

Responsible Disclosure with Ledger On May 2017 we contacted the Ledger's security team to inform them about our findings. We exchanged a series of emails in which we described our attacks and suggested our aforementioned fix. The Ledger team aknowledged the attacks and stated that they were planning to fix their implementation. Our communication ended on the 9th of May 2017. On June 2017 Ledger paused all sales of both Nano and Nano S until the end of August, stating on their website that it was due to high demand. On late August, Ledger re-introduced in the market only Nano S and released a new firmware which encrypts the low-level communication. Although Ledger has provided a fix, all devices running the old firmware are still vulnerable, as the update is not automatic.[7]

[7]On December 2017 we tested our devices to check whether the update is enforced automatically. Our findings showed that no automatic firmware update occurs.

Chapter 11

Conclusion

The purpose of this thesis is to shed light on the vulnerabilities of the low-level implementations of cryptographic protocols. We studied the communication layer of smart-cards with respect to the high-level protocol they implement. In particular, we analysed the implementations on PKCS#11 and Bitcoin smart-cards. The goal of such cards is to provide guarantees against tampering and to offer an isolated environment for executing sensitive operations. Our work has proven that by focussing on the low-level communication layer one can mount attacks that are not possible at higher layers, rendering smart-cards vulnerable, and, thereby defeating the purpose of using them as tamper-resistant environments.

Summary of contributions. A common practice among PKCS#11 smart-cards vendors is to define their own proprietary implementations and intentionally hide all details. Manually analysing and testing such cards is a time consuming process that requires complex technical skills and a very deep understanding of the communication standards. To that end, we provide a solution for automatically extracting the implementation details of the communication layer. REPROVE can infer the semantics of the exchanged commands as well as the on-card operations in a matter of milliseconds. We used REPROVE to reverse-engineer the implementation of seven commercially available smart-cards. The output of REPROVE is generally useful to identify deviations from the standard and implementation flaws. But it is primarily useful because it allows us to identify security vulnerabilities and threats in the implementation of the low-level communication protocol.

To analyse the security of the extracted models we defined a threat model for PKCS#11 smart-cards with respect to the communication layer. The purpose is to identify as many as possible attack scenarios with respect to a malicious adversary's goal. Since the mitigation of each attack is dependent on the exact implementation, it requires refinement to each card's characteristics. Instead of specifying individual attack strategies for each card, we identified patterns between the attacks to navigate the threat search space. We converged to a general set of implementation-independent vulnerabilities. We used that set as a guide to analyse the extracted models of the smart-cards. As REPROVE's extracted models suggested, most of the cards do not enforce a secure channel; the authentication is stateless and the communication is in plaintext. Consequently, such implementations are prone to a set of attacks. Among other findings, the most significant one was the API-side execution of cryptographic functions after a plaintext key transmission. This is a major vulnerability that defeats the purpose of incorporating cryptographic hardware into a system. Another surprising finding was that user private data was not protected; the messages to be signed and encrypted, and the output of the decryption, were sent in plaintext, which enabled replay and tampering attacks. These are just sample of the discovered vulnerabilities, which are all intentionally hidden behind the proprietary implementations that the vendors define. REPROVE's outputs can be conducive for understanding the card's behaviour.

Although there exist other systems that aim at extracting patterns of a card's implementations, they do require the semantics of the implementation to be known beforehand. We show how REPROVE can be integrated with such systems and provide useful input. We presented an ecosystem for extracting meaningful state-machines of a card's implementation. The results provide a deeper understanding of the card's behaviour under unexpected input and present command sequences that lead to vulnerabilities. To that end we demonstrated practical attacks on one of the tested cards. We show that the keys' attributes are not protected. This allowed us to tamper with these attributes and change the roles of a key with respect to the cryptographic functions that use it. Moreover, we demonstrated how keys can be redirected by changing the handle of the key. Such attacks could be prevented if accessing this data was protected by a stateful authentication mechanism or if the PKCS#11 policies were hardcoded on the card.

Securing the low-level communication is not only a matter of PKCS#11 smart-cards. Other applications that incorporate smart-cards may be prone to similar threats. To test that assumption, we analysed the Bitcoin smart-card wallets that are available on the market. Bitcoin is particularly popular amongst users that seek to perform fast, secure and private transactions. Opting for a hardware wallet such as smart-cards to enhance the security of the transactions has become a common practice. We show that even an expensive smart-card wallet that promises a secure and isolated environment for the management of the account is prone to low-level attacks.

We extracted and analysed the low-level protocols that the LEDGER wallets implement. The LEDGER protocols are customised to the Bitcoin specifications and do not allow the execution of irrelevant operations. The communication throughout an active session proceeds in plaintext and is not protected by any mechanism *e.g.*, session keys, session counter, mutual authentication between the hardware and the API *etc.* This allowed us to identify a series of vulnerabilities that ultimately led to loss of the account's ownership. To our surprise, we discovered a plaintext transmission of the master secret key. In Bitcoin, this can easily escalate to possession of the entire wallet and consequently of the account. Other exploits that we discovered include the plaintext transmission of encryption keys and the lack of protection of the transaction data. To that end, we demonstrated practical attacks that take advantage of such vulnerabilities. Our attacks range from eavesdropping sensitive information, to disabling the security mechanisms of the wallet and tampering with the processed transactions. Finally, we proposed possible fixes which include interaction with the user, to prevent such attacks.

Outlook. The purpose of offering a secure implementation should not only focus on the protection of the cryptographic key values but also on the computational outcome of the cryptographic operations. As we presented in this thesis it is possible to tamper with that outcome without knowledge of the key's value. Another aspect, heavily overlooked, is the user's privacy. As in all security protocols, the APDU layer should protect all transmitted private data as well as data that can expose the user's identity. This property is particularly important not only for the Bitcoin, but for the PKCS#11 smart-cards as well. Protection mechanisms that provide solutions to such problems exist; ISO 7816 specifies mechanisms for secure messaging, which can randomise parts of or whole commands. Unfortunately, such mechanisms are not enforced by the vendors. One reason for that might be that they prefer speed over security, as adding extra cryptographic operations may bring delays to the system. The lack of well defined threat models definitely contributes to bad implementation decisions which can be exploited.

As security is seen as the measure against an adversary gaining access to sensitive components of a system, isolating these components is considered to be best practice. Hardware technology has evolved and offers dedicated devices specifically designed to fulfil this criterion and it is more than common for systems to incorporate such hardware for operations over sensitive data. In the meantime, hardware security has grown into an active research area, offering methods for ensuring physical security spanning from verification tools to countermeasures against attacks. Most importantly, hardware can be certified on its security levels by dedicated laboratories. Such certification occurs after testing the hardware for vulnerabilities; if no attacks can be mounted then the hardware passes the tests. The guarantees that hardware may bring into a system have made them a popular solution for adding extra security. However, security is commonly evaluated mainly by focussing on cryptography, overlooking logical attacks that can occur either on the hardware's software or interacting protocols. The work presented in this thesis focuses on how protocols interact with the hardware.

Combining a provably secure protocol with certified tamper-resistant hardware devices is considered enough to guarantee the security of a system. However, isolation of the sensitive components is not enough on its own. The devices being distinctive features of a system can make them a direct target. Systems that are advertised to be secure, are in practice vulnerable because of logical flaws and false assumptions hidden behind secret, error-prone implementations.

Security through obscurity is a common practice among smart-card vendors. The overall message of this thesis is that although hiding the implementation details may provide some levels of security, it can certainly not guarantee it. Security should be considered on the whole system, not only on idividual layers. Adding secure components without carefully designing their interaction ultimately opens unpredictable attack vectors. All these can be prevented if there existed well-defined and verified standards and specifications that the vendors could based their implementations (interindustry or proprietary) on. Currently there is a gap between low-level implementations and security, not only in smart-cards but in cryptographic hardware in general. As the technology evolves, the different threats evolve as well. Static, secret, unverified implementations cannot cope with that evolution. A good practice would be making the specifications open to the public to allow discussions and feedback from the research community. We hope this thesis is helpful to the discussion of the need for open and verifiable standards and implementations for end-to-end security provisioning.

Appendix A

PKCS#11: REPROVE Background Knowledge

Commands.

```
    command(_INS, 'e0', _P1, P2, _Lc, D, _Le, List, NewList, _Response):- add(isa(P2, offset), List, NList),
add(create_file(D), NList, NewList).
    command(_INS,'e4', 0, _P2, _Lc, D, _Le, List, NewList, _Response):- add(isa(D, df),List, A),
add(delete(file, D), A, NewList).
    command(_INS,'e4', 4, _P2, _Lc, D, _Le, List, NewList, _Response):- add(isa(D, df),List, A),
add(delete(file, D), A, NewList).
    command(_INS,'e4', 2, _P2, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, ef), List, A),
add(delete(file, D ), A, NewList).
    command(_INS,'e4', 8, _P2, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, df), List, A),
add(delete(file, D ), A, NewList).
    command(_INS,'e4', 9,_P2, _Lc, D, _Le, List, NewList, _Response):-
add(delete(path, D), A, NewList).
    command(_INS,'e4', _P1, 09,_Lc, D, _Le, List, NewList, _Response):-
add(isa(D, df), List, A),
add(delete(path, D), A , NewList).
    command(_INS,'e4', P1,_P2, _Lc, D, _Le, List, NewList, _Response):-
between(128,160,P1),
add(isa(D, ef), List, A),
add(delete(file, D), A, NewList).
    command(_INS,'e4', _, _,0,0,null, List, NewList, _Response):-
add(dummy(null), List, NewList).
    command(_,'22',1, _P2, Lc, D, _Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',17, _P2, Lc, D, _Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',33, _P2, Lc, D, _Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',49, _P2, Lc, D, _Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',97, _P2, Lc, D,_Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',113, _P2, Lc, D, _Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',225, _P2, Lc, D, _Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',241, _P2, Lc, D, _Le, List, NewList, Response):-
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',P1, _P2, Lc, D, _Le, List, NewList, Response):-
between(32, 47, P1),
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(set, D, Response), AL, NewList).
    command(_,'22',P1, _P2, _Lc, D, null, List, NewList, _Response):-
between(16, 31, P1),
add(isa(D, secureMes), List, AL),
add(manage_security_environment(secureMessaging, D, noResponse), AL, NewList).
    command(_,'22',P1, _P2, _Lc, D, _Le, List, NewList, Response):-
between(48, 63, P1),
add(isa(D, secureMes), List, AL),
add(isa(Response, secureMes), AL, AL),
add(manage_security_environment(secureMessaging, D, Response), AL, NewList).
    command(_,'22',P1, _P2, _Lc, D, null, List, NewList, _Response):-
between(80, 127, P1),
add(isa(D, secureMes), List, AL),
add(manage_security_environment(internalCardCryptoOperation, D, noResponse), AL, NewList).
    command(_,'22',P1, _P2, 0, 0, _Le, List, NewList, Response):-
between(160, 175, P1),
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(externalCardCryptoOperation, noD, Response), AL, NewList).
    command(_,'22',P1, _P2, 0, 0, _Le, List, NewList, Response):-
between(144, 159, P1),
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(externalCardCryptoOperation, noD, Response), AL, NewList).
    command(_,'22',P1, _P2, 0, 0, _Le, List, NewList, Response):-
between(224, 241, P1),
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(externalCardCryptoOperation, noD, Response), AL, NewList).
    command(_,'22',P1, _P2, 0, 0, _Le, List, NewList, Response):-
between(244, 255, P1),
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(externalCardCryptoOperation, noD, Response), AL, NewList).
    command(_,'22',P1, _P2, 0, 0, _Le, List, NewList, Response):-
between(242, 243, P1),
add(isa(Response, secureMes), List, AL),
add(isa(Response, key), AL, AL),
add(manage_security_environment(storeOperationKey, noD, Response), AL, NewList).
    command(_,'22',P1, _P2, 0, 0, _Le, List, NewList, Response):-
between(242, 243, P1),
add(isa(Response, secureMes), List, AL),
add(manage_security_environment(internalCardOperation, noD, Response), AL, NewList).
    command(_, '42',255 ,255, Lc, D, Le, List, NewList, Response):-
add(isa(D, rfu), List, AL),
add(tag(D,rfu), AL, AL),
add(isa(Response, rfu),AL, AAL),
add(tag(Response, rfu), AAL, AAAL),
add(security_operation(D, Response), AAAL, NewList).
    command(_, '42',142 ,80, Lc, D, Le, List, NewList, Response):-
add(isa(D, originalData), List, AL),
add(isa(Response, checksum),AL, AL),
add(crypto_checksum(D, Response), AL, NewList).
    command(_, '42',158 ,154, Lc, D, null, List, NewList, _Response):-
add(isa(D, dataToSign), List, AL),
add(isa(D, data),AL, AL),
add(isa(Response, digitalSignature), AL, AAL),
add(digital_signature(D, null), AAL, NewList).
    command(_, '42',158 ,172, Lc, D, null, List, NewList, _Response):-
add(isa(D, valuesToSign), List, AL),
add(isa(D, dataObjects),AL, AL),
add(isa(Response, digitalSignature), AL, AAL),
add(digital_signature(D, null), AAL, NewList).
    command(_, '42',158 ,179, Lc, D, _Le, List, NewList, Response):-
add(isa(D, dataToSign), List, AL),
add(isa(D, dataObjects),AL, AL),
add(isa(Response, digitalSignature), AL, AAL),
add(digital_signature(D, null), AAL, NewList).
    command(_, '42',158 ,154, Lc, D, Le, List, NewList, Response):-
add(isa(D, dataToSign), List, AL),
add(isa(D, data),AL, AL),
add(isa(Response, digitalSignature), AL, AAL),
add(digital_signature(D, Response), AAL, NewList).
    command(_,'42' ,158 ,172, Lc, D, Le, List, NewList, Response):-
add(isa(D, valuesToSign), List, AL),
add(isa(D, dataObjects),AL, AL),
add(isa(Response, digitalSignature), AL, AAL),
add(digital_signature(D, Response), AAL, NewList).
    command(_, '42',158 ,188, Lc, D, Le, List, NewList, Response):-
add(isa(D, dataToSign), List, AL),
add(isa(D, dataObjects),AL, AL),
add(isa(Response, digitalSignature), AL, AAL),
add(digital_signature(D, Response), AAL, NewList).
    command(_, '42',00 ,168, Lc, D, Le, List, NewList, Response):-
add(isa(D, signature), List, AL),
add(verify_signature(D), AL, NewList).
    command(_, '42',255 ,P2, Lc, D, Le, List, NewList, Response):-
add(isa(rfu, tag), List, AL),
add(tag(D, rfu), AL, AL),
add(isa(P2, tag),AL, AAL),
add(tag(Response, P2), AAL, AAAL),
add(security_operation(D, Response), AAAL, NewList).
    command(_, '42',00 ,P2, 00, 00, Le, List, NewList, Response):-
add(isa(null, tag), List, AL),
add(tag(D,null), AL, AL),
add(isa(P2, tag),AL, AAL),
add(tag(Response, P2), AAL, AAAL),
add(security_operation(null, Response), AAAL, NewList).
    command(_, '42',P1 ,255, Lc, D, Le, List, NewList, Response):-
add(isa(P1, tag), List, AL),
add(tag(D,P1), AL, AL),
add(isa(rfu, tag),AL, AAL),
add(tag(Response, rfu), AAL, AAAAL),
add(security_operation(D, Response), AAAL, NewList).
    command(_, '42',P1 ,00, Lc, D, null, List, NewList, _Response):-
add(isa(P1,tag ), List, AL),
add(tag(D,P1), AL, AL),
add(isa(null, tag),AL, AAL),
add(tag(Response, null), AAL, AAAAL),
add(security_operation(D, Response), AAAAL, NewList).
    command(_, '42',P1 ,P2, Lc, D, null, List, NewList, Response):-
add(isa(P1, tag), List, AL),
add(tag(D, P1), AL, AL),
add(isa(P2, tag),AL, AAL),
add(tag(noResponse, P2), AAL, AAAL),
add(security_operation(D, noResponse), AAAL, NewList).
    command(_, '42',P1 ,P2, Lc, D, Le, List, NewList, Response):-
add(isa(P1, tag), List, AL),
add(tag(D, P1), AL, AL),
add(isa(P2, tag),AL, AAL),
add(tag(Response, P2), AAL, AAAL),
add(security_operation(D, Response), AAAL, NewList).
    command(_INS,'a4', _, _,0,0,null, List, NewList, _Response):-
add(dummy(null), List, NewList).
    command(_INS,'a4', 0, _P2, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, df),List, A),
add(select(file, D), A, NewList).
    command(_INS,'a4', 4, _P2, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, df),List, A),
add(select(file, D), A, NewList).
    command(_INS,'a4', 2, _P2, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, ef), List, A),
add(select(file, D ), A, NewList).
    command(_INS,'a4', 8, _P2, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, df), List, A),
add(select(file, D ), A, NewList).
    command(_INS,'a4', 9,_P2, _Lc, D, _Le, List, NewList, _Response):-
add(select(path, D), A, NewList).
    command(_INS,'a4', _P1, 09,_Lc, D, _Le, List, NewList, _Response):-
add(isa(D, df), List, A),
add(select(path, D), A , NewList).
    command(_INS,'a4', P1,_P2, _Lc, D, _Le, List, NewList, _Response):-
between(128,160,P1),
add(isa(D, ef), List, A),
add(select(file, D), A, NewList).
    command(_, ’44’, 0,0,0,0,null, List, NewList,_Response):-
add(active(current), List, A),
add(activate(Type, F), A, NewList).
    command(_, ’44’, 0, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df),List, A),
add(activate(D), A,B),
add(activate(file, D), B, NewList).
    command(_, ’44’, 4, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(active(D), A, B),
add(activate(file, D), B, NewList).
    command(_, ’44’, 2, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, ef), List, A),
add(active(D), A, B),
add(activate(file, D), B, NewList).
    command(_,’44’, 8, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(active(D), A, B),
add(activate(path, D), B, NewList).
    command(_,’44’, 9, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(active(D), A, B),
add(activate(path, D), B, NewList).
    command(_,’44’, _P1, 09,_Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(active(D), A, B),
add(activate(path, D), B, NewList).
    command(_,’44’,P1, _P2, _Lc, D, null, List, NewList, _Response):-
between(128,160,P1),
add(isa(D, ef), List, A),
add(active(D), A, B),
add(activate(file, D), B, NewList).
    command(_INS, ’ca’, 01, 0, 0, 0, Le, List, NewNewList, Response):-
add(isa(Response, cardByteString), List, NewList),
add(retrieve_get_data(Le, cardByteString, Response), NewList, NewNewList).
    command(_INS, ’ca’, 255, 255, 0, 0, Le, List, NewList, Response):-
add(isa(Response, simple_tlv), List, A),
add(retrieve_get_data(Le, simple_tlv, Response), A, NewList).
    command(_INS, ’ca’, 02, 0, 0, 0, Le, List, NewList, Response):-
add(isa(Response, rfu), List, A),
add(retrieve_get_data(Le, rfu, Response), A, NewList).
    command(_INS, ’ca’, 0, P2, 0, 0, Le, List, NewList, Response):-
between(40,255, P2),
add(isa(Response, ber_tlv), List, A),
add(retrieve_get_data(Le, ber_tlv, Response), A, NewList).
    command(_INS, ’ca’, 02, P2, 0, 0, Le, List, NewList, Response):-
between(0,255, P2),
add(isa(Response, simple_tlv), List, A),
add(retrieve_get_data(Le, simple_tlv, Response), A, NewList).
    command(_INS, ’ca’, P1,P2, 0, 0, Le, List, NewList, Response):-
between(40, 255, P1),
between(0,255, P2),
add(isa(Response, rfu), List, A),
add(retrieve_get_data(Le, rfu, Response), A, NewList).
    command(_INS, 'cb', 255, 255, _Lc, _D, Le, List, ANewList, Response):-
add(isa(Response, dataObjectlist), List, NewList),
add(retrieve_get_data(Le, list, Response), NewList, ANewList).
    command(_INS, 'dd', 00, 55, _Lc,D, _Le, List, NewList, _Response):-
add(isa, (D, record), List, AL),
add(isa(D, offset), AL, BList),
add(write_data(D, D), BList, NewList).
    command(_INS, 'dc', P1, P2, _Lc,D, _Le, List, NewList, _Response):-
between(274, 512, P2),
add(isa(P2, ef), List, AL),
add(select(file, P2), AL, BList),
add(isa(D, record), [], CList),
add(isa(P1, offset), CList, DList),
add(write_data(D, P1), DList, NewDList),
add(BList, [], A),
add(NewDList, A, NewList).
    command(_INS, 'dc', P1, _P2, _Lc,D, _Le, List, NewList, _Response):-
add(isa(D,record), List, AL),
add(isa(P1, offset), AL, BList),
add(write_data(D, P1), BList, NewList).
    command(_INS, 'b0', P1, P2, 0, 0, Le, List, NewList, Response):-
between(128, 160, P1),
add(isa(P1, ef), List, A),
add(select(file, P1), A, LA),
add(isa(P2, offset), [], L),
add(isa(Response, binary), L, LL),
add(retrieve_data(Le, binary, Response), LL, LLL),
add(LA, [], NewLA),
add(LLL, NewLA, NewList).
    command(_INS, 'b0', P1, P2, 0, 0, Le, List, NewList, Response):-
between(0, 128, P1),
add(isa([P1,P2], offset), List, NList),
add(isa(Response, binary), NList, NNList),
add(retrieve_data(Le, binary, Response), NNList, NewList).
    command(_INS, 'b1', P1, P2, _Lc, _D, Le, List, NewList, Response):-
add(isa([P1,P2], ef), List, A),
add(select(file, [P1,P2]), A, LA),
add(isa(P2, offset), [], L),
add(isa(Response, binary), L, LL),
add(retrieve_data(Le, binary, Response), LL, LLL),
add(LA, [], NewLA),
add(LLL, NewLA, NewList).
    command(_INS, 'b2', P1, _P2,0, 0, Le, List, NewList, Response):-
add(isa(P1, offset), List, AL),
add(isa(Response, record), AL, A),
add(retrieve_data(Le, data, Response), A, NewList).
    command(_INS, 'b3', _P1, _P2, _Lc, D, Le, List, NewList, Response):-
add(isa(D, offset), List, NList),
add(isa(Response, record), NList, NNList),
add(retrieve_data(Le, data, Response), NNList, NewList).
    command(_INS, 'db', 0, P2, _Lc, D, null, List, NewList, _Response):-
```

```
between(40,255, P2),
add(isa(D, ber_tlv), List, A),
add(write_data(card, D, FiLe), A, NewList).
        command(_INS, ’db’, 0, P2, _Lc, D, null, List, NewList, _Response):-
between(0,63, P2),
add(isa(D, rfu), List, A),
add(write_data(card, D, FiLe), A, NewList).
        command(_INS, ’db’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,255, P2),
between(40,255, P1),
add(isa(D, ber_tlv), List, A),
add(write_data(card, D, FiLe), A, NewList).
        command(_INS, ’da’, 0, P2, _Lc, D, null, List, NewList, _Response):-
between(40,255, P2),
add(isa(D, ber_tlv), List, A),
add(write_data(card, D, FiLe), A, NewList).
        command(_INS, ’db’, 0, P2, _Lc, D, null, List, NewList, _Response):-
between(0,63, P2),
add(isa(D, rfu), List, A),
add(write_data(card, D, FiLe), A, NewList).
        command(_INS, ’db’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,255, P2),
between(40,255, P1),
add(isa(D, ber_tlv), List, A),
add(write_data(card, D, FiLe), A, NewList).
        command(_INS, ’d0’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128, 160, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(write_data(D, P2), A, NewList).
        command(_INS, ’d0’,P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0, 127,P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(write_data(D, P1/¶2), A, NewList).
        command(_INS, ’d1’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128, 160, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(write_data(D, P2), A, NewList).
        command(_INS, ’d1’,P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0, 127,P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(write_data(D, P1/¶2), A, NewList).
        command(_INS, ’d6’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(64,127, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(write_data(D, P2), A, NewList).
        command(_INS, ’d6’, 00, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(write_data(D, P1/¶2), A, NewList).
    command(_INS, ’d6’, P1, P2, _Lc, D, null, List, NewList, _Response):-
member(P1, [1,3,19,17,33,35,49,51,65,67,81,85,97,99,113,115,129,131,154,147,161,163,177,179,193,195,209,211,225,227,241,243]),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(write_data(D, P2), A, NewList).
    command(_INS, ’d6’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(write_data(D, P1/¶2), A, NewList).
    command(_INS, ’d6’, 01, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, data), List, AL),
add(isa(D, offset), AL, NList),
add(write_data(D, D), NList, NewList).
    command(_INS, ’d7’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(64,127, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(write_data(D, P2), A, NewList).
    command(_INS, ’d7’, 00, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(write_data(D, P1/¶2), A, NewList).
    command(_INS, ’d7’, P1, P2, _Lc, D, null, List, NewList, _Response):-
member(P1, [1,3,19,17,33,35,49,51,65,67,81,85,97,99,113,115,129,131,154,147,161,163,177,179,193,195,209,211,225,227,241,243]),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(write_data(D, P2), A, NewList).
    command(_INS, ’d7’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(write_data(D, P1/¶2), A, NewList).
    command(_INS, ’d7’, 01, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, data), List, AL),
add(isa(D, offset), AL, NList),
add(write_data(D, D), NList, NewList).
    command(_INS, ’0c’, 00, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa, (D, record), List, AL),
add(isa(D, offset), AL, BList),
add(erase_data(D, D), BList, NewList).
    command(_INS, ’46’, P1, 00, _Lc, D, null, List, NewList, null):-
generateList([0], RangeList, 2, 256,1),
member(P1, RangeList),
add(isa(op, keyPair), List, AL),
add(isa(null, noResponse), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’47’, P1, 00, _Lc, D, null, List, NewList, null):-
generateList([0], RangeList, 2, 256,1),
member(P1, RangeList),
add(isa(op, keyPair), List, AL),
add(isa(null, noResponse), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’46’, P1, 00, _Lc, D, null, List, NewList, null):-
generateList([1], RangeList, 2, 253,1),
member(P1, RangeList),
add(isa(op, publicKey), List, AL),
add(isa(null, noResponse), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’47’, P1, 00, _Lc, D, null, List, NewList, null):-
generateList([1], RangeList, 2, 253,1),
member(P1, RangeList),
add(isa(op, publicKey), List, AL),
add(isa(null, noResponse), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’46’, 00, 00, _Lc, D, null, List, NewList, Response):-
add(isa(op,unknown), List, AL),
add(isa(Response, unknown), AL, AL),
add(generate_asymmetricKeyPair, AL, NewList).
    command(_INS, ’47’, 00, 00, _Lc, D, null, List, NewList, Response):-
add(isa(op,unknown), List, AL),
add(isa(Response, unknown), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’46’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([0], RangeList, 2, 256,1),
member(P1, RangeList),
add(isa(op, keyPair), List, AL),
add(isa(Response, pk), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList)
    command(_INS, ’47’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([0], RangeList, 2, 256,1),
member(P1, RangeList),
add(isa(op, keyPair), List, AL),
add(isa(Response, pk), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList)
    command(_INS, ’46’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([1], RangeList, 2, 253,1),
member(P1, RangeList),
add(isa(op, publicKey), List, AL),
add(isa(Response, pk), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’47’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([1], RangeList, 2, 253,1),
member(P1, RangeList),
add(isa(op, publicKey), List, AL),
add(isa(Response, pk), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’46’,P1, 00, _Lc, D, null, List, NewList, Response):-
add(isa(op,unknown), List, AL),
add(isa(Response, unknown), AL, AL),
add(generate_asymmetricKeyPair, AL, NewList).
    command(_INS, ’47’, 00, 00, _Lc, D, null, List, NewList, Response):-
add(isa(op,unknown), List, AL),
add(isa(Response, unknown), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’46’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([0], RangeList, 2, 256,1),
member(P1, RangeList),
generateList([12], RangeListA, 8, 256,1), \+member(P1, RangeListA),
add(isa(op, keyPair), List, AL),
add(isa(Response, unknown), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList)
    command(_INS, ’47’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([0], RangeList, 2, 256,1),
member(P1, RangeList),
generateList([12], RangeListA, 8, 256,1), \+member(P1, RangeListA),
add(isa(op, keyPair), List, AL),
add(isa(Response, unknown), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList)
    command(_INS, ’46’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([1], RangeList, 2, 253,1),
member(P1, RangeList),
generateList([12], RangeListA, 8, 256,1), \+member(P1, RangeListA),
add(isa(op, publicKey), List, AL),
add(isa(Response, unknown), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’47’, P1, 00, _Lc, D, null, List, NewList, Response):-
generateList([1], RangeList, 2, 253,1),
member(P1, RangeList),
generateList([12], RangeListA, 8, 256,1), \+member(P1, RangeListA),
add(isa(op, publicKey), List, AL),
add(isa(Response, unknown), AL, AAL),
add(generate_asymmetricKeyPair, AAL, NewList).
    command(_INS, ’e2’, 00, 00, _Lc, D, null, List, NewList, _Response):-
add(active(current), List, AL),
add(isa(D, record), AL, BList),
add(append_data(D, D), BList, NewList).
    command(_INS, ’e2’, 00, P2, _Lc, D, null, List, NewList, _Response):-
add(active(P2), List, AL),
add(isa(D, record), AL, BList),
add(append_data(D, D), BList, NewList).
add(write_data(card, D, FiLe), A, NewList).
    command(_INS, ’d2’, 0, P2, _Lc, D, null, List, NewList, _Response):-
between(40,255, P2),
add(isa(D, record), List,AL),
add(isa(currentOffset, offSet), AL, A),
add(write_data(D, currentOffset), A, NewList).
    command(_INS, ’d2’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,7, P2),
add(isa(D, record), List, AL),
add(isa(currentOffset, offset), AL, A),
add(write_data(D, currentOffset), A, NewList).
    command(_INS, ’d2’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
generateList([0], RangeList, 8, 248, 1),
member(P2, RangeList),
add(isa(D, record), List, A),
add(isa(firstOffset, offset), A, AL),
add(write_data(D, firstOffset), AL, NewList).
        command(_INS, 'd2', _P1, P2, _Lc, D, null, List, NewList, _Response):-
generateList([2], RangeList, 8, 250,1),
member(P2, RangeList),
add(isa(D, record), List, A),
add(isa(nextOffset, offset), A, AL),
add(write_data(D, nextOffset), AL, NewList).
        command(_INS, 'd2', P1, P2, _Lc, D, 0,List, NewList, _Response):-
generateList([4], RangeList, 8, 252,1),
member(P2, RangeList),
add(isa(D, record), List, A),
add(isa(P1, offset), A, AL),
add(write_data(D, P1), AL,NewList).
        command(_, 'e4', 0,0,0,0,null, List, NewList,_Response):-
add(deactive(current), List, A),
add(deactivate(Type, F), A, NewList).
        command(_, 'e4', 0, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df),List, A),
add(deactivate(D), A,B),
add(deactivate(file, D), B, NewList).
        command(_, 'e4', 4, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(deactive(D), A, B),
add(deactivate(file, D), B, NewList).
        command(_, 'e4', 2, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, ef), List, A),
add(deactive(D), A, B),
add(deactivate(file, D), B, NewList).
        command(_,'e4', 8, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(deactive(D), A, B),
add(deactivate(path, D), B, NewList).
        command(_,'e4', 9, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(deactive(D), A, B),
add(deactivate(path, D), B, NewList).
        command(_,'e4', _P1, 09,_Lc, D, null, List, NewList, _Response):-
add(isa(D, df), List, A),
add(deactive(D), A, B),
add(deactivate(path, D), B, NewList).
        command(_,'e4',P1, _P2, _Lc, D, null, List, NewList, _Response):-
between(128,160,P1),
add(isa(D, ef), List, A),
add(deactive(D), A, B),
add(deactivate(file, D), B, NewList).
        command(_INS, 'c0', 0, 0, 0, 0, Le, List, NewList, Response):-
add(get_response(Le,Response), List, NewList).
        command(_INS, '84', 0, 0, 0, 0, Le, List, NewList, Response):-
add(get_challenge(Le, Response), List, NewList).
        command(_INS, '0e', P1, P2, _Lc, D, null, List, NewList, _Response):-
between(64,127, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, '0e', 00, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, '0e', P1, P2, _Lc, D, null, List, NewList, _Response):-
member(P1, [1,3,19,17,33,35,49,51,65,67,81,85,97,99,113,115,129,131,154,147,161,163,177,179,193,195,209,211,225,227,241,243]),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, '0e', P1, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, '0e', 01, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, data), List, AL),
add(isa(D, offset), AL, NList),
add(erase_data(D, D), NList, NewList). command(_INS, '0e', P1, P2, _Lc, D, null, List, NewList, _Response):-
between(64,127, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, '0e', 00, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, '0e', P1, P2, _Lc, D, null, List, NewList, _Response):-
member(P1, [1,3,19,17,33,35,49,51,65,67,81,85,97,99,113,115,129,131,154,147,161,163,177,179,193,195,209,211,225,227,241,243]),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, '0e', P1, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, '0e', 01, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, data), List, AL),
add(isa(D, offset), AL, NList),
add(erase_data(D, D), NList, NewList).
    command(_INS, '0f', P1, P2, _Lc, D, null, List, NewList, _Response):-
between(64,127, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, '0f', 00, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, ’0f’, P1, P2, _Lc, D, null, List, NewList, _Response):-
member(P1, [1,3,19,17,33,35,49,51,65,67,81,85,97,99,113,115,129,131,154,147,161,163,177,179,193,195,209,211,225,227,241,243]),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, ’0f’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, ’0f’, 01, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, data), List, AL),
add(isa(D, offset), AL, NList),
add(erase_data(D, D), NList, NewList).
    command(_INS, ’0f’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(64,127, P1),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, ’0f’, 00, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, ’0f’, P1, P2, _Lc, D, null, List, NewList, _Response):-
member(P1, [1,3,19,17,33,35,49,51,65,67,81,85,97,99,113,115,129,131,154,147,161,163,177,179,193,195,209,211,225,227,241,243]),
add(isa(D, string), List, AL),
add(isa(P2, offset), AL, A),
add(erase_data(D, P2), A, NewList).
    command(_INS, ’0f’, P1, P2, _Lc, D, null, List, NewList, _Response):-
between(192,256, P1),
add(isa(D, string), List, AL),
add(isa(P1/¶2, offset), AL, A),
add(erase_data(D, P1/¶2), A, NewList).
    command(_INS, ’0f’, 01, _P2, _Lc, D, null, List, NewList, _Response):-
add(isa(D, data), List, AL),
add(isa(D, offset), AL, NList),
add(erase_data(D, D), NList, NewList).
    command(_INS, ’20’, 0, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown), List, A),
add(verify(D), A,NewList).
    command(_INS, ’20’, 0, P2, _Lc, D, _Le, List, NewList, _Response):-
between(0,127, P2),
add(isa(D, global), List, N),
add(verify(D), N, NewList).
    command(_INS, ’20’, 0, P2, _Lc, D, _Le, List, NewList, _Response):-
between(128,256, P2),
add(isa(D, specific), List, A),
add(verify(D), A, NewList).
    command(_INS, ’21’, 0, P2, _Lc, D, _Le, List, NewList, _Response):-
between(128,256, P2),
add(isa(D, specific), List, L),
add(verify(D), L, NewList).
    command(_INS, '21', 0, P2, _Lc, D, _Le, List, NewList, _Response):-
between(0,127, P2),
add(isa(D, global), List, L),
add(verify(D), L, NewList).
    command(_INS, '21', 0, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown), List,A),
add(verify(D), A, NewList).
    command(_INS, '82', _P1, 0, _Lc, D, null, List, NewList, _Response):-
add(isa(D, unknown), List, A),
add(external_authenticate(D), A,NewList).
    command(_INS, '82', _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, '82', _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128,256,P2),
add(isa(D, specific), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, '86', _P1, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, '86', _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, 86, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128,256,P2),
add(isa(D, specific), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, '87', _P1, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, '87', _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, '87', _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128,256,P2),
add(isa(D, specific), List,A),
add(external_authenticate(D), A,NewList).
    command(_INS, '86', _P1, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown), List,A),
add(internal_authenticate(D),A, NewList).
    command(_INS, '86', _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List,A),
add(internal_authenticate(D), A,NewList).
    command(_INS, '86', _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128,256,P2),
add(isa(D, specific), List, A),
add(internal_authenticate(D), A, NewList).
    command(_INS, '87', _P1, 0, _Lc, D, _Le, List, NewList, _Response):-
```

```
add(isa(D, unknown), List, A),
add(internal_authenticate(D),A, NewList).
        command(_INS, ’87’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List, A),
add(internal_authenticate(D), A,NewList).
        command(_INS, ’87’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128,256,P2),
add(isa(D, specific),List,A),
add(internal_authenticate(D), A,NewList).
        command(_INS, ’86’, _P1, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown), List, A),
add(mutual_authenticate(D), A,NewList).
        command(_INS, ’86’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List, A),
add(mutual_authenticate(D), A,NewList).
        command(_INS, ’86’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128,256,P2),
add(isa(D, specific), List,A),
add(mutual_authenticate(D), A,NewList).
        command(_INS, ’87’, _P1, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown), List,A),
add(mutual_authenticate(D), A,NewList).
        command(_INS, ’87’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global),List, A),
add(mutual_authenticate(D), A,NewList).
        command(_INS, ’87’, _P1, P2, _Lc, D, null, List, NewList, _Response):-
between(128,256,P2),
add(isa(D, specific), List, A),
add(mutual_authenticate(D), A,NewList).
        command(_INS, ’88’, _P1, 0, _Lc, D, _Le, List, NewList, _Response):-
add(isa(D, unknown),List, A),
add(internal_authenticate(D), A,NewList).
        command(_INS, ’88’, _P1, P2, _Lc, D, _Le, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List, A),
add(internal_authenticate(D), A, NewList).
        command(_INS, ’88’, _P1, P2, _Lc, D, _Le, List, NewList, _Response):-
between(128,256,P2),
(isa(D, specific), List, A),
add(internal_authenticate(D), A,NewList).
        command(_INS, ’88’, _P1, P2, _Lc, D, _Le, List, NewList, _Response):-
between(0,127,P2),
add(isa(D, global), List,A),
add(internal_authenticate(D), A,NewList).
```

Functionalities.

```
        functionality(authentication(TypeA, TypeB, Sensitivity), [secret_verify(TypeA, TypeB)],
[secret_verify(TypeA, TypeB)], Sensitivity).
```

functionality(authentication(Challenge, Response, Sensitivity), [challenge_sent(Le, Challenge), external_authenticate(Response,TypeOfResponse)], [challenge_sent(Le, Challenge), external_authenticate(Response,TypeOfResponse)], Sensitivity).

functionality(authentication(Challenge, Response, Sensitivity), [read_data_sub(card, Le, Challenge), external_authenticate(Response, TypeOfResponse)], [read_data_sub(card, Le, Challenge), external_authenticate(Response, TypeOfResponse)], Sensitivity).

functionality(authentication(TypeA, TypeB), [internal_authenticate(TypeA, TypeB)], _).

functionality(authentication(TypeA, TypeB), [mutual_authenticate(TypeA, TypeB)], _).

functionality(store_data(Location, D, Sensitivity), [file_create(Location, D)], [file_create(Location, D)], Sensitivity).

functionality(store_data(Location, D, Sensitivity), [data_write(Location, D)], [data_write(Location, D)], Sensitivity).

functionality(read_data(Location, Le, RD, Sensitivity), [read_data_sub(Location,Le, RD)], [read_data_sub(Location,Le, RD)], Sensitivity).

functionality(sign(Data, Response, Sensitivity), [perform_digital_signature(Data, Response)], [perform_digital_signature(Data, Response)],Sensitivity).

functionality(sign(Data, Response, Sensitivity), [isa(EF, ef),select(file, EF), perform_digital_signature(Data, Response)], [perform_digital_
signature(Data, Response)], Sensitivity).

functionality(sign(Data, Response, Sensitivity), [read_data_sub(EF, Le, Key), perform_digital_signature(Data, Response)], [read_data_sub(EF, Le, Key), perform_digital_signature(Data, Response)], Sensitivity).

functionality(sign(Data, Response, Sensitivity), [isa(EF, ef), select(file, EF), perform_digital_signature(Data, null), read_data_sub(card, Le, Response)], [perform_digital_signature(Data, null), read_data_sub(card, Le, Response)], Sensitivity).

functionality(sign(C,RD, Sensitivity),[security_environment(Operation, D1, D2), perform_operation(C,B), read_data(Location, Le, RD)], [perform_operation(C,B), read_data(Location, Le, RD)], Sensitivity).

functionality(sign(Data, Response, Sensitivity), [isa(EF, ef), select(file, EF), security_environment(Operation, D1,D2), perform_digital_signature(Data, Response)], [perform_digital_signature(Data, Response)], Sensitivity).

functionality(sign(Data, Response, Sensitivity),[read_data_sub(EF, Le, Key), security_environment(Operation, D1,D2), perform_digital_signature(Data, Response)], [perform_digital_signature(Data, Response)], Sensitivity).

functionality(sign(Data, Response, Sensitivity), [isa(EF, ef), select(file, EF), security_environment(Operation, D1,D2), perform_digital_signature(Data, null), read_data_sub(card, Le, Response)], [read_data_sub(EF, Le, Key), perform_digital_signature(Data, null), read_data_sub(card, Le, Response)], Sensitivity).

functionality(verify(ResponseSignature, RD), [verify_digital_signature(ResponseSignature, RD)], _,_).

functionality(generateKey(Response), [generate(keyPair,Response), _,_].

Sub-functionalities.

subfunctionality(read_data_sub(EF, Le, RD), [[isa(EF, ef),select(file, EF)], [isa(O,offset), isa(RD, DataType),retrieve_data(Le, Data, RD)]], [retrieve_data(Le, Data, RD)]).

subfunctionality(read_data_sub(DF, Le, RD), [[isa(DF, df),select(file, DF)], [isa(RD, DataType),retrieve_get_data(Le, Data, RD)]], [retrieve_get_data(Le, Data, RD)]).

subfunctionality(read_data_sub(DF, Le, RD), [[isa(DF, df),select(file, DF)], [isa(N, offset), isa(RD, DataType),retrieve_data(Le, Data, RD)]], [retrieve_data(Le, Data, RD)]).

subfunctionality(read_data_sub(DF, Le, RD), [[isa(RD, DataType),retrieve_get_
data(Le, Data, RD)]], [retrieve_data(Le, DataType, RD)]).

subfunctionality(read_data_sub(DF, Le, RD),[[isa(DF, offset), isa(RD, record), retrieve_data(Le, DataType, RD)]], [retrieve_data(Le, DataType, RD)]).

subfunctionality(read_data_sub(card, Le, RD), [[get_response(Le, RD)]], [get_response(Le, RD)], get_response(Le, RD)).

subfunctionality(data_write(Location, D), [[isa(Location, ef),select(file, Location)], [isa(D, DataType), isa(Offset, offset), write_data(D, Offset)]], [write_data(D, Offset)]).

subfunctionality(data_write(Location, D), [[isa(Location, df),select(file, Location)], [isa(Offset, offset), write_data(D, Offset)]], [write_data(D, Offset)]).

subfunctionality(data_write(card, Location, D), [[isa(Location, df),select(path, Location)], [isa(D, DataType), write_data(D, Location)]], [write_data(D, Location)]).

subfunctionality(data_write(card, Location, D), [[isa(Location, ef),select(file, Location)], [isa(D, DataType), write_data(D, Location)]], [write_data(D, Location)],_).

subfunctionality(data_write(card, Location, D), [[isa(File, ef),select(file, Location)], [isa(D, DataType), write_data(card, D, Location)]], [write_data(card,D,Location)],_).

subfunctionality(data_write(Offset, D), [[isa(D, DataType), isa(Offset, offset), write_data(D, Offset)]], _, [write_data(D, Offset)]).

subfunctionality(data_write(Offset, D), [[isa(Offset, offset), write_data(D, Offset)]], [write_data(D, Offset)],_).

subfunctionality(data_write(card, null, D), [[isa(D, DataType), write_data(D, Location)]], [write_data(D, Location)],_).

subfunctionality(data_write(Location, D), [[write_data(D, null), _]],[write_data(D, null)],_).

subfunctionality(file_create(DF, D), [[isa(DF, df), select(path,DF)], [isa(O,offset),create_file(D)]], [create_file(D)],_).

subfunctionality(file_create(DF, D), [[isa(DF, df), select(file,DF)],[isa(O,offset), create_file(D)]], [create_file(D)],_).

subfunctionality(file_delete(DF, D), [[isa(D, df), select(path,DF)], [isa(D, df),delete_file(file, D)]], [delete_file(_,D)]).

subfunctionality(file_delete(DF, D), [[isa(D, ef), select(file,DF)], [isa(D, ef),delete_file(file, D)]], [delete_file(_,D)]).

subfunctionality(file_delete(DF, D), [[isa(D, df), select(path,DF)], [isa(D, df),delete_file(path, D)]], [delete_file(_,D)]).

subfunctionality(file_create(DF, D), [[isa(DF, df), select(file,DF)],[isa(O,offset), delete_file(D)]], [delete_file(D)],_).

subfunctionality(file_activate(Type, File), [[isa(File, Type), select(FP, File)], [active(current), activate(FP, File)]], [activate(FP, File)],_).

subfunctionality(file_activate(Type, File), [[isa(File, Type)], [active(File), activate(FP, File)]], [activate(FP, File)],_).

subfunctionality(generate(Op,Response), [[isa(op, OP), isa(Response, Type), generate_asymmetricKeyPair], [generate_asymmetricKeyPair]]).

subfunctionality(challenge_sent(Le, Challenge), [[get_challenge(Le, Challenge)]], [get_challenge(Le, Challenge)],_).

subfunctionality(challenge_sent(Le, Challenge), [[isa(File, Type),select(Kind, File)], [get_challenge(Le, Challenge)]], [get_challenge(Le, Challenge)],_).

subfunctionality(secret_verify(pin, D), [[isa(D, TypeOfPasscode), verify(D)]], _, verify(D)).

subfunctionality(secret_verify(pin, D), [[isa(EF, ef),select(file, EF)], [isa(D, TypeOfPasscode), verify(D)]], _, [verify(D)]).

subfunctionality(secret_verify(pin, D), [[isa(DF, df),select(file, DF)], [isa(D, TypeOfPasscode), verify(D)]], _, [verify(D)]).

subfunctionality(secret_verify(pin, D), [[isa(DF, df),select(path, DF)], [isa(D, TypeOfPasscode), verify(D)]], _, [verify(D)]).

subfunctionality(external_authenticate(Response, TypeOfResponse),[[isa(Response, TypeOfResponse),external_authenticate(Response)]] ,_, [external_authenticate(Response)]).

subfunctionality(external_authenticate(Response, TypeOfResponse),[[isa(File, Type), select(Kind, File)], [isa(Response, TypeOfResponse),external_authenticate(Response)]] ,_, [external_authenticate(Response)]).

subfunctionality(internal_authenticate(Response, TypeOfResponse),[[isa(Response, TypeOfResponse),internal_authenticate(Response)]] ,_, [internal_authenticate(Response)]).

subfunctionality(internal_authenticate(Response, TypeOfResponse),[[isa(File, Type), select(Kind, File)], [isa(Response, TypeOfResponse),internal_authenticate(Response)]] ,_, [internal_authenticate(Response)]).

subfunctionality(perform_digital_signature(Data, Response),[[isa(Data, dataToSign), isa(Data, Type), isa(Response, digitalSignature), digital_signature(Data, Response)]], [digital_signature(Data, Response)],_).

subfunctionality(perform_digital_signature(Data, Response),[[isa(P, tag), tag(Data, P), isa(RTag, tag), tag(Response, RTag), security_operation(Data, Response)]],[security_ operation(Data, Response)],_).

subfunctionality(verify_digital_signature(Data), [[verify_signature(Data)]], _,_).

subfunctionality(verify_digital_signature(Data, Response), [[isa(P, tag), tag(Data, P), isa(RTag, tag), tag(Response, RTag), security_operation(Data, Response)]], [security_operation(Data, Response)],_).

subfunctionality(hash(C,B),[[security_operation_hash(B,C)]],[security_operation_ hash(B,C)],_).

subfunctionality(decrypt(C,B),[[security_operation_decrypt(B,C)]],[security_operation_ decrypt(B,C)],_).

subfunctionality(encrypt(C,B),[[security_operation_encrypt(B,C)]],[security_operation_ encrypt(B,C)],_).

subfunctionality(security_environment(Operation, D1, D2), [[isa(D2, Meaning2), manage_security_environment(Operation, D1, D2)]],[manage_security_ environment(Operation, D1, D2)],_).

subfunctionality(security_environment(Operation, D1, D2), [[isa(D1, Meaning1), isa(D2, Meaning2), manage_security_environment(Operation, D1, D2)]],[manage_security_ environment(Operation, D1, D2)],_).

subfunctionality(security_environment(Operation, D1, D2), [[isa(D1, Meaning1), manage_security_environment(Operation, D1, D2)]],[manage_security_ environment(Operation, D1, D2)],_).

PKCS#11 Models.

pkcs(log_in, [authentication(A,B, _directSensitive)]).
pkcs(log_in, [read_data(Location, File, RD, _indirectSensitive), authentication(A,B,_directSensitive)]).
pkcs(generate_key, [generateKey(Response)]).
pkcs(generate_key,[store_data(Location, D,S),generateKey(Response)]).
pkcs(generate_key, [authentication(A,B, _directSensitive),generateKey(Response)]).
pkcs(generate_key,[authentication(A,B, _directSensitive), store_data(Location, D,S),generateKey(Response)]).
pkcs(sign, [part_sign(D, Rd, S)]).
pkcs(find_objects, [read_data(Location, Le, RD,S)]).
pkcs(get_attributeValue, [read_data(Location, Le, RD, S)]).
pkcs(get_attributeValue, [read_data(Location, Le, RD, S), store_data(Location, D,S)]).
pkcs(set_attributeValue,[store_data(Location, D, S)]).
pkcs(set_attributeValue,[store_data(File, FileData,S), store_data(File, D, S)]).
pkcs(set_attributeValue,[read_data(Location, Le, RD,S), store_data(Location, D, S)]).
pkcs(wrap_key,[read_data(Location, Le,RD, S)]).
pkcs(wrap_key,[encrypt(C,B]).
pkcs(encrypt,[encrypt(C,B)]).

Appendix B

PKCS#11: Extracted State-Machines

B.1 Cardos V4.3, V4.4, V5 State-Machines

Although Cardos V4.3 was produced by a different vendor (Siemens) from V4.4 and V5 (Atos), all tested smart-cards have the exact same implementation. The state-machines are presented in Figure B.2, Figure B.3, Figure B.4, and Figure **??**.

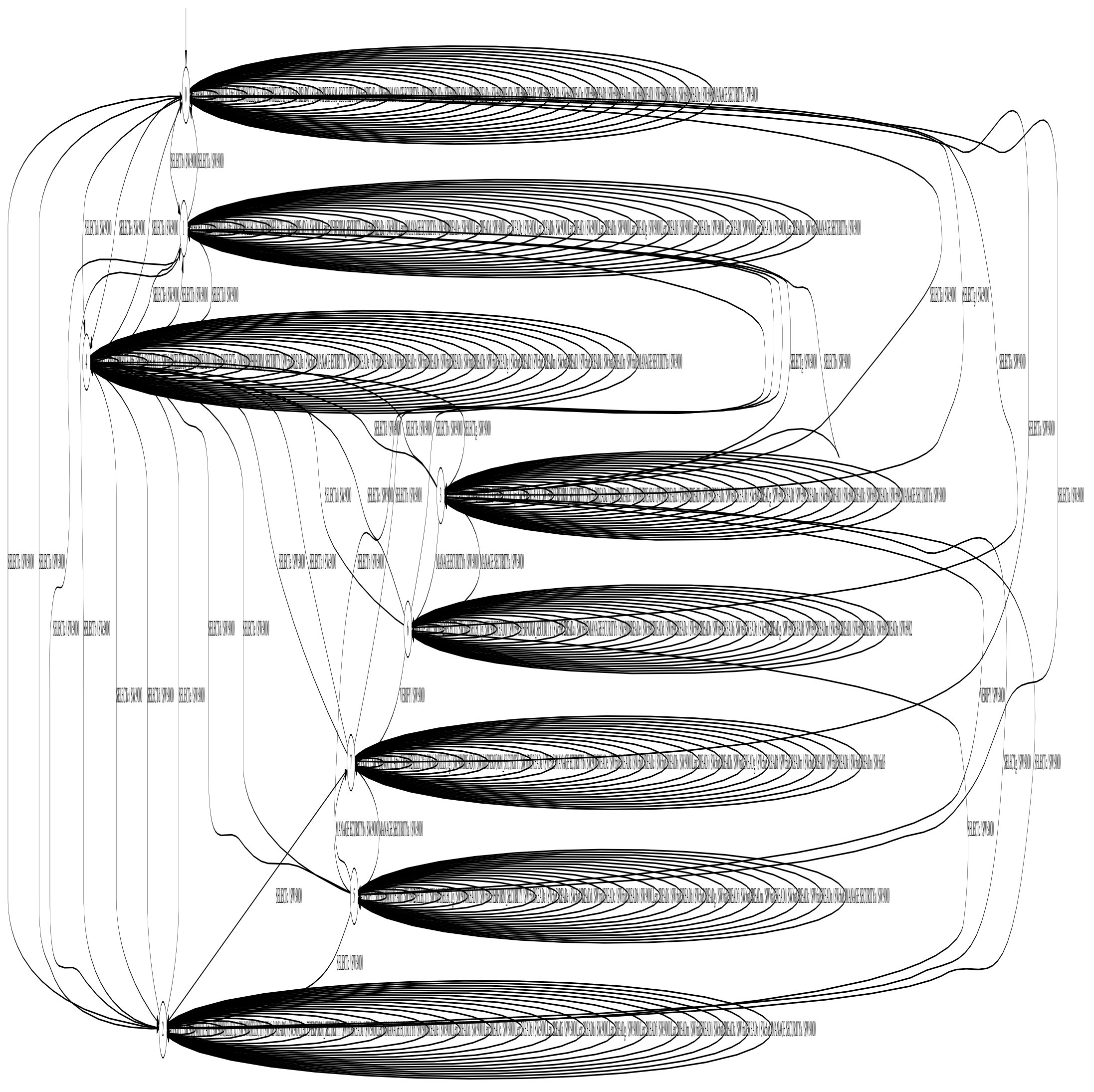


Figure B.1: The extracted state machine of the Cardos V4.4 for the functions `C_logIn`, `C_getAttributeValue`, `C_setAttributeValue`, `C_sign` and `C_findObjects` functions.

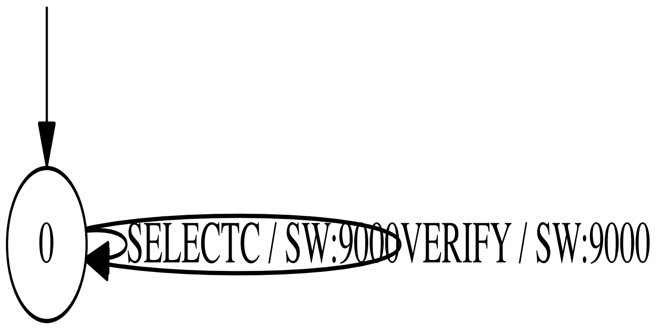


Figure B.2: The extracted state-machine of the `C_logIn` function for the Cardos smart-cards.

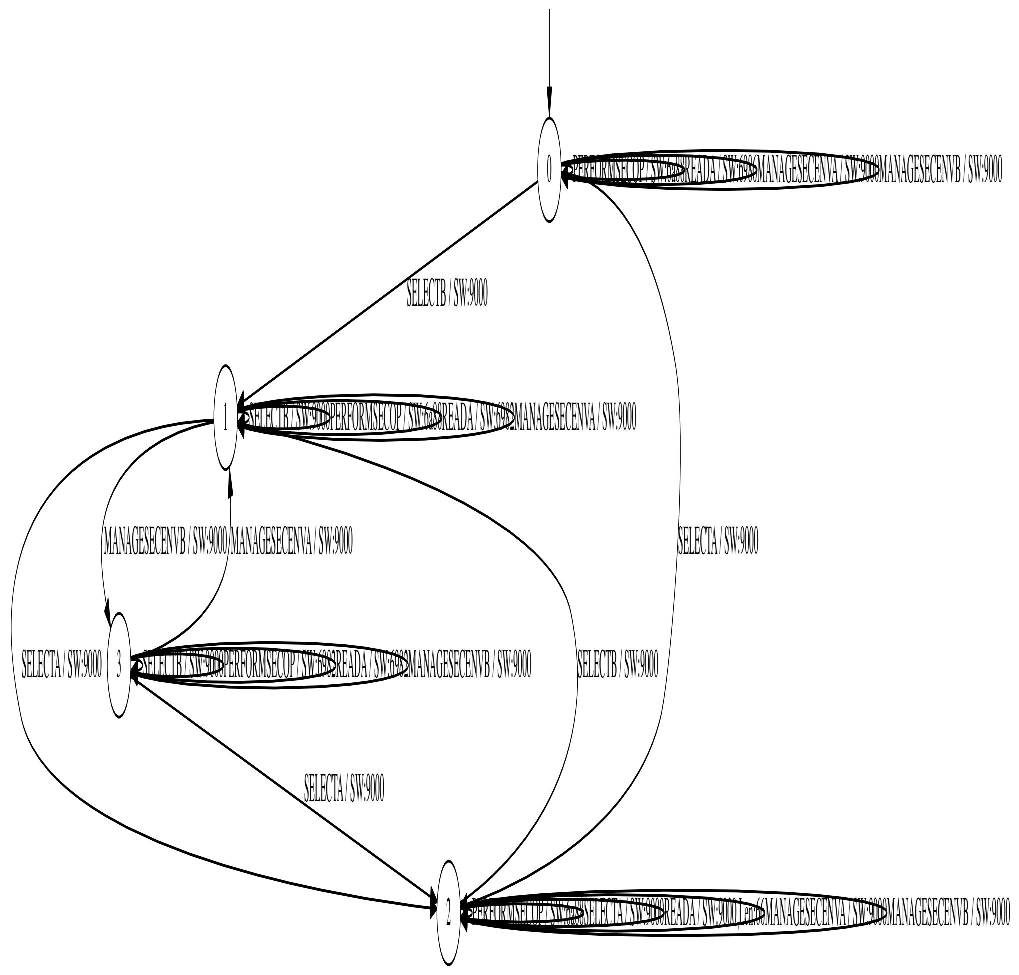

0
PERFORMSECOP / SW:6a88READA / SW:6986MANAGESECENVA / SW:9000MANAGESECENVB / SW:9000
SELECTB / SW:9000
1
SELECTB / SW:9000PERFORMSECOP / SW:6a88READA / SW:6982MANAGESECENVA / SW:9000
MANAGESECENVB / SW:9000
MANAGESECENVA / SW:9000
SELECTA / SW:9000
SELECTA / SW:9000
3
SELECTB / SW:9000PERFORMSECOP / SW:6982READA / SW:6982MANAGESECENVB / SW:9000
SELECTB / SW:9000
SELECTA / SW:9000
2
PERFORMSECOP / SW:6a88SELECTA / SW:9000READA / SW:9000 Len:66MANAGESECENVA / SW:9000MANAGESECENVB / SW:9000

Figure B.3: The extracted state-machine of the `C_sign` function, without authentication, for the Cardos smart-cards.

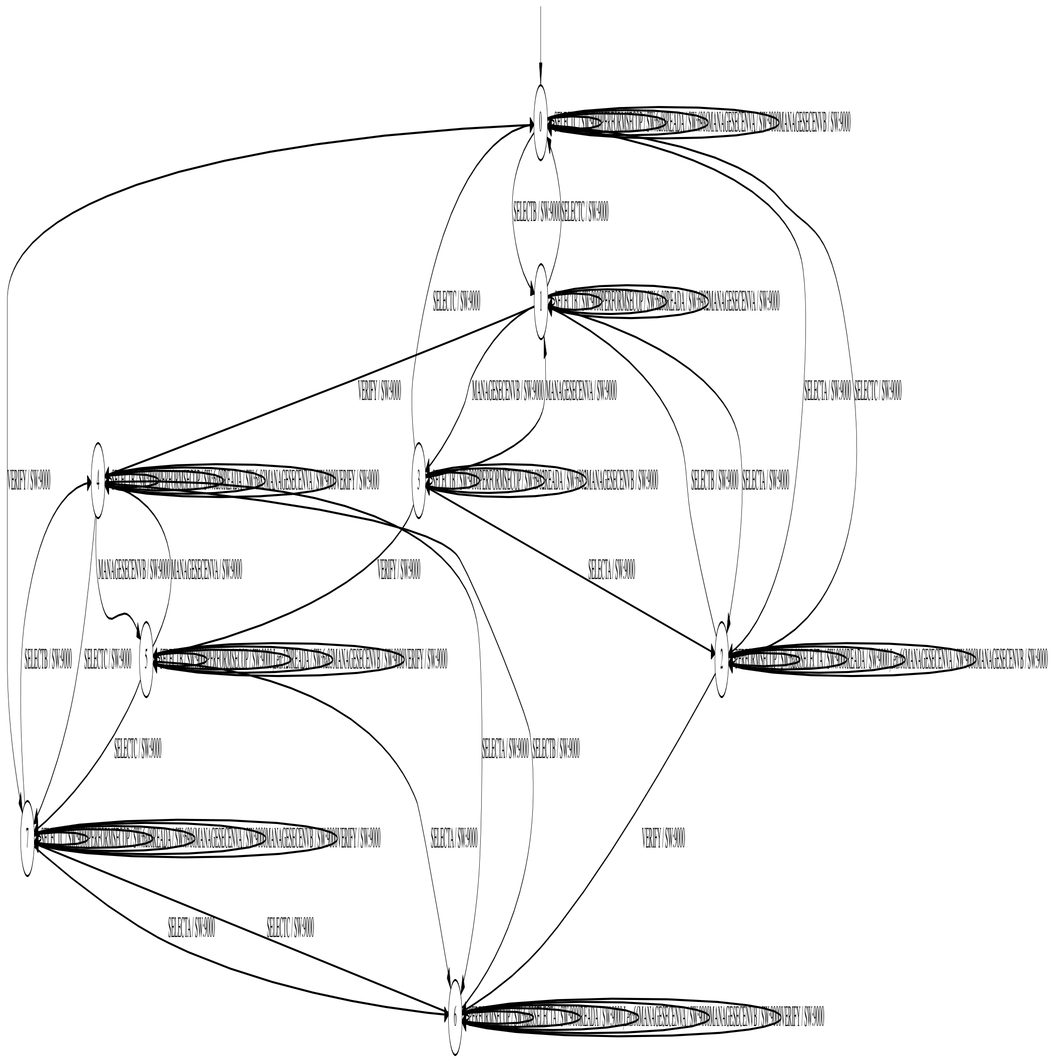


Figure B.4: The extracted state-machine of the C_sign function including authentication, for the Cardos smart-cards

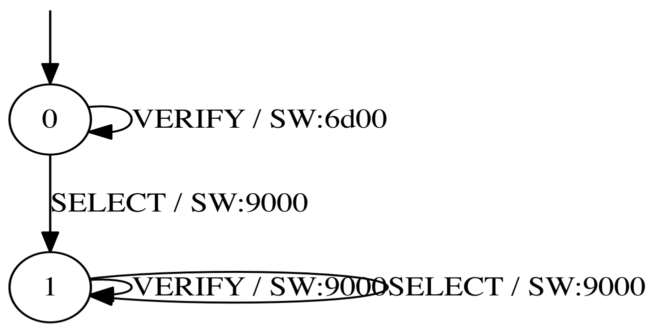


Figure B.6: The extracted state-machine of the `C_login` function for the Safesite smart-card.

B.2 Safesite TCP IS V1

The extracted state-machines for the Safesite TCP IS V1 smart-card are presented in Figure B.6, Figure B.7, Figure B.8, Figure B.9 and Figure B.10.

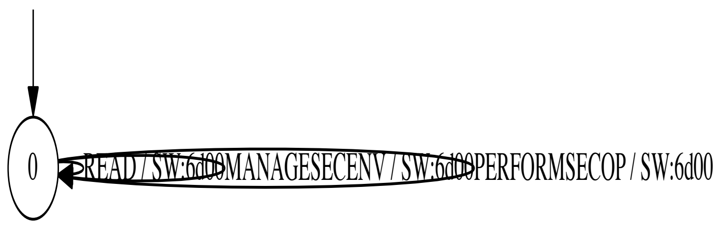


Figure B.7: The extracted state-machine of the `C_sign` function without authentication, for the Safesite smart-card.

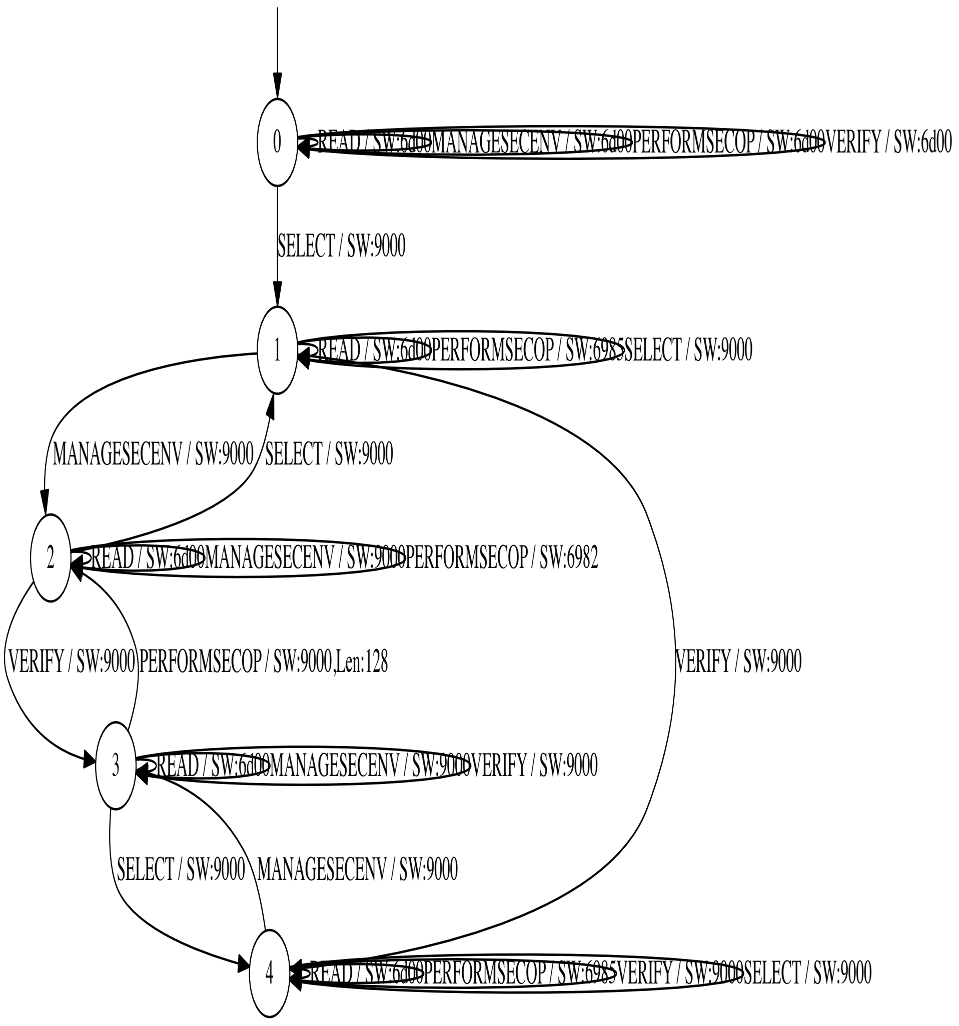


Figure B.8: The extracted state-machine of the `C_sign` function including authentication, for the Safesite smart-card.

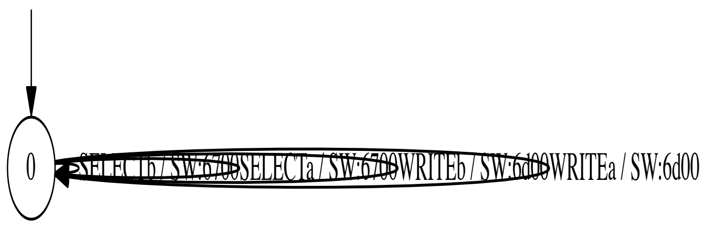


Figure B.9: The `C_setAttributeValue`, without authentication, state-machine of the Safesite smart-card

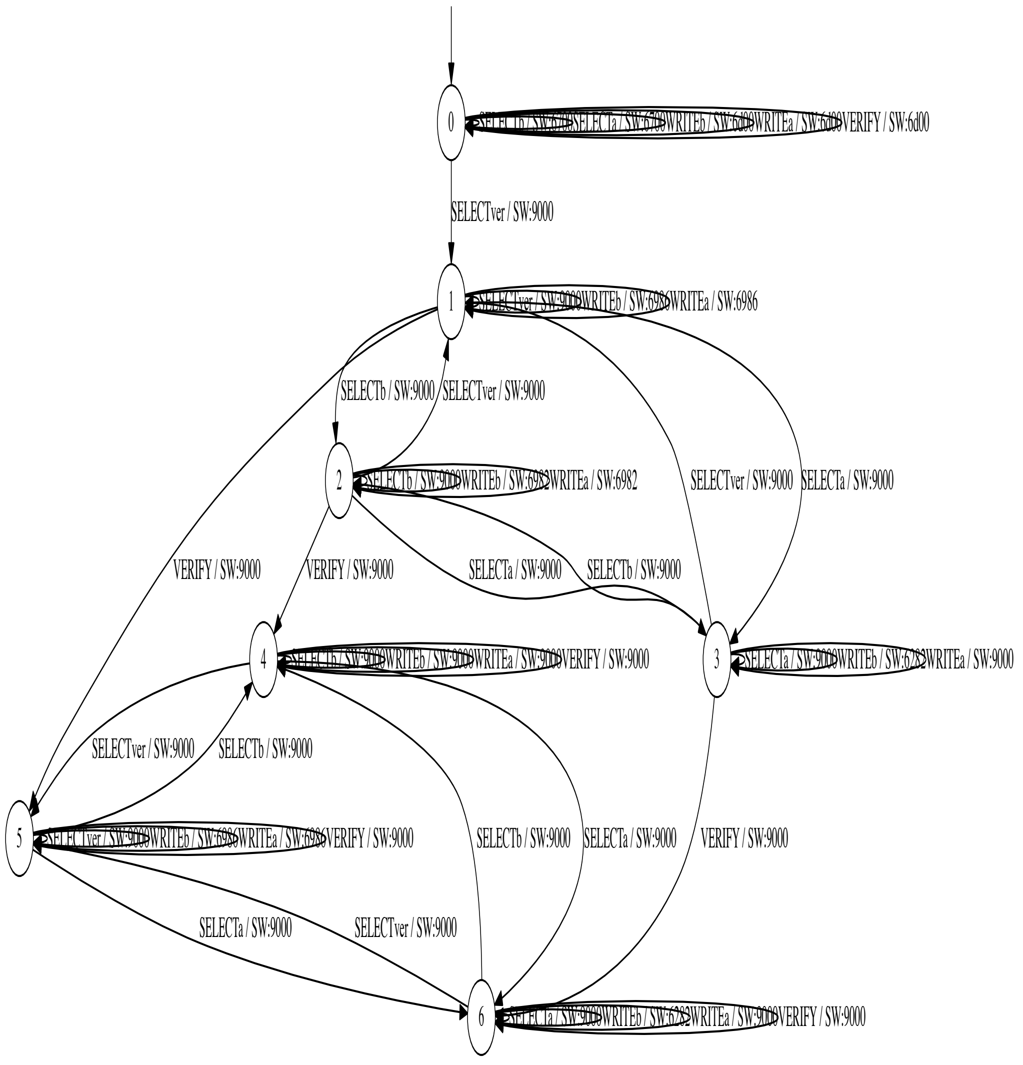


Figure B.10: The `C_setAttributeValue` state-machine, including authentication, for the Safesite smart-card

Bibliography

[Anderson et al., 2006] Anderson, R., Bond, M., and Murdoch, S. J. (2006). Chip and spin. *Computer Security Journal*, 22(2):1–6.

[Anderson and Kuhn, 1996] Anderson, R. and Kuhn, M. (1996). Tamper resistance-a cautionary note. In *USENIX Workshop on Electronic Commerce*, volume 2, pages 1–11.

[Androulaki et al., 2013] Androulaki, E., Karame, G., Roeschlin, M., Scherer, T., and Capkun, S. (2013). Evaluating user privacy in bitcoin. In *Financial Cryptography and Data Security International Conference, Revised Selected Papers*, pages 34–51.

[Bamert et al., 2014] Bamert, T., Decker, C., Wattenhofer, R., and Welten, S. (2014). Bluewallet: The secure bitcoin wallet. In *International Workshop Security and Trust*, pages 65–80.

[Bar-El et al., 2004] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., and Whelan, C. (2004). The sorcerer's apprentice guide to fault attacks. *Cryptology ePrint Archive*.

[Barber et al., 2012] Barber, S., Boyen, X., Shi, E., and Uzun, E. (2012). Bitter to better - how to make bitcoin a better currency. In *Financial Cryptography and Data Security International Conference, Revised Selected Papers*, pages 399–414.

[Barbu et al., 2011] Barbu, G., Duc, G., and Hoogvorst, P. (2011). Java card operand stack: Fault attacks, combined attacks and countermeasures. In *Smart Card Research and Advanced Application Conference*, pages 297–313.

[Barbu et al., 2012] Barbu, G., Giraud, C., and Guerin, V. (2012). Embedded eavesdropping on java card. In *Information Security and Privacy Research*, pages 37–48.

[Benadjila et al., 2014] Benadjila, R., Calderon, T., and Daubignard, M. (2014). Caml crush: A PKCS#11 filtering proxy. In *Smart Card Research and Advanced Applications*, pages 173–192. Springer.

[Biham and Shamir, 1997] Biham, E. and Shamir, A. (1997). Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference*, pages 513–525.

[Bitcoin, 2016] Bitcoin (2016). Segregated Witness:The Next Steps.

[Bitcoinmagazine, 2013] Bitcoinmagazine (2013). Ozcoin stolen funds. *https: // goo. gl/ WmcG2n.*

[Bond and Anderson, 2001] Bond, M. and Anderson, R. (2001). Api-level attacks on embedded systems. *Computer*, 34(10):67–75.

[Bond et al., 2004] Bond, M., Cvrček, D., and Murdoch, S. J. (2004). Unwrapping the Chrysalis. Technical Report UCAM-CL-TR-592, University of Cambridge, Computer Laboratory.

[Bortolozzo et al., 2010] Bortolozzo, M., Centenaro, M., Focardi, R., and Steel, G. (2010). Attacking and fixing PKCS#11 security tokens. In *ACM Conference on Computer and Communications Security*, pages 260–269.

[Bozzato et al., 2016] Bozzato, C., Focardi, R., Palmarini, F., and Steel, G. (2016). APDU-level attacks in PKCS# 11 devices. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 97–117.

[Caballero et al., 2007] Caballero, J., Yin, H., Liang, Z., and Song, D. (2007). Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*, pages 317–329.

[Centenaro et al., 2012] Centenaro, M., Focardi, R., and Luccio, F. L. (2012). Type-based analysis of PKCS# 11 key management. In *International Conference on Principles of Security and Trust*, pages 349–368. Springer.

[Cho et al., 2010] Cho, C. Y., Babi ć, D., Shin, E. C. R., and Song, D. (2010). Inference and analysis of formal models of botnet command and control protocols. In *ACM Conference on Computer and Communications Security*, pages 426–439.

[Choudary, 2010] Choudary, O. (2010). The Smart Card Detective: a hand-held emv interceptor, University of Cambridge, Computer Laboratory, Darwin College, MPhil thesis.

[Clulow, 2003] Clulow, J. (2003). On the Security of PKCS#11. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 411–425.

[Comparetti et al., 2009] Comparetti, P. M., Wondracek, G., Kruegel, C., and Kirda, E. (2009). Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125.

[Cortier et al., 2007] Cortier, V., Keighren, G., and Steel, G. (2007). Automatic analysis of the security of xor-based key management schemes. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 538–552.

[Courant and Monin, 2006a] Courant, J. and Monin, J.-F. (2006a). Defending the Bank with a Proof assistant. In *International Workshop on Issues in the Theory of Security*, pages 87–98.

[Courant and Monin, 2006b] Courant, J. and Monin, J.-F. (2006b). Defending the bank with a proof assistant. In *International Workshop on Issues in the Theory of Security*.

[Cryptosense, 2014] Cryptosense (2014). Grand jury indictment of pleschuck. *goo.gl/YkfFrQ*.

[Cui et al., 2007] Cui, W., Kannan, J., and Wang, H. J. (2007). Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*.

[D. Longley, 1992] D. Longley, S. R. (1992). An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1).

[De Cock et al., 2005] De Cock, D., Wouters, K., Schellekens, D., Singelee, D., and Preneel, B. (2005). Threat modelling for security tokens in web applications. In *Communications and Multimedia Security*, pages 183–193. Springer.

[De Koning Gans and de Ruiter, 2012] De Koning Gans, G. and de Ruiter, J. (2012). The SmartLogic Tool: Analysing and testing smart card protocols. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 864–871.

[Decker and Wattenhofer, 2014] Decker, C. and Wattenhofer, R. (2014). Bitcoin transaction malleability and mtgox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer.

[Delaune et al., 2008] Delaune, S., Kremer, S., and Steel, G. (2008). Formal analysis of PKCS#11. In *Computer Security Foundations*, pages 331–344.

[Delaune et al., 2010] Delaune, S., Kremer, S., and Steel, G. (2010). Formal security analysis of PKCS# 11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245.

[Diffie and Hellman, 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

[DigiNotar, 2014] DigiNotar (2014). Black Tulip Report. `cryptosense.com/wp-content/uploads/2014/11/black-tulip-update.pdf`.

[Dolev and Yao, 1983] Dolev, D. and Yao, A. (1983). On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208.

[Fides Aarts, 2013] Fides Aarts, Joeri de Ruiter, E. P. (2013). Formal models of bank cards for free. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, volume 0, pages 461–468.

[Fröschle and Sommer, 2011] Fröschle, S. and Sommer, N. (2011). Concepts and proofs for configuring PKCS#11. In *International Workshop on Formal Aspects in Security and Trust*, pages 131–147. Springer.

[Gandolfi et al., 2001] Gandolfi, K., Mourtel, C., and Olivier, F. (2001). Electromagnetic analysis: Concrete results. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 251–261. Springer.

[Genkin et al., 2014] Genkin, D., Shamir, A., and Tromer, E. (2014). RSA key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference*, pages 444–461.

[Gkaniatsou et al., 2015] Gkaniatsou, A., McNeill, F., Bundy, A., Steel, G., Focardi, R., and Bozzato, C. (2015). Getting to know your card: reverse-engineering the smart-card application protocol data unit. In *ACM Annual Computer Security Applications Conference*, pages 441–450.

[Google,] Google. Protocol Buffers. `developers.google.com/protocol-buffers`.

[Hao and Ryan, 2010] Hao, F. and Ryan, P. (2010). *J-PAKE: Authenticated Key Exchange without PKI*, pages 192–206. Springer.

[Herrera-Joancomartí, 2014] Herrera-Joancomartí, J. (2014). Research and challenges on bitcoin anonymity. In *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance - 9th International Workshop, DPM 2014, 7th International Workshop, SETOP 2014, and 3rd International Workshop, QASA 2014. Revised Selected Papers*, pages 3–16.

[Higgins, 2015] Higgins, S. (2015). Bitstamp bitcoin echange. `www.coindesk.com/unconfirmed-report-5-million-bitstamp-bitcoin-exchange`.

[Hsiao et al., 2009] Hsiao, H.-C., Lin, Y.-H., Studer, A., Studer, C., Wang, K.-H., Kikuchi, H., Perrig, A., Sun, H.-M., and Yang, B.-Y. (2009). A study of user-friendly hash comparison schemes. In *ACM Annual Computer Security Applications Conference*, pages 105–114.

[Huang et al., 2014] Huang, D. Y., Dharmdasani, H., Meiklejohn, S., Dave, V., Grier, C., McCoy, D., Savage, S., Weaver, N., Snoeren, A. C., and Levchenko, K. (2014). Botcoin: Monetizing stolen cycles. In *Annual Network and Distributed System Security Symposium*.

[Iguchi-Cartigny and Lanet, 2010] Iguchi-Cartigny, J. and Lanet, J.-L. (2010). Developing a trojan applet in a smart card. *Journal in computer virology*, 6(4):343–351.

[Iso.org, 1198] Iso.org (1198). ISO/IEC 7816-1:1998 Identification cards - Integrated circuit(s) cards with contacts Part 1: Physical characteristics.

[Iso.org, 2005a] Iso.org (2005a). ISO/IEC 7816-10:1999 Identification cards - Integrated circuit(s) cards with contacts Part 10: Electronic signals and answer to reset for synchronous cards.

[Iso.org, 2005b] Iso.org (2005b). ISO/IEC 7816-11:2004 Identification cards - Integrated circuit cards Part 11: Personal verification through biometric methods.

[Iso.org, 2005c] Iso.org (2005c). ISO/IEC 7816-12:2005 Identification cards - Integrated circuit cards Part 12: Cards with contacts USB electrical interface and operating procedures.

[Iso.org, 2005d] Iso.org (2005d). ISO/IEC 7816-13:2007 Identification cards - Integrated circuit cards Part 13: Commands for application management in a multi-application environment.

[Iso.org, 2005e] Iso.org (2005e). ISO/IEC 7816-15:2004 Identification cards - Integrated circuit cards Part 15: Cryptographic information application.

[Iso.org, 2005f] Iso.org (2005f). ISO/IEC 7816-4:2005 Identification cards - Integrated circuit cards Part 4: Organization, security and commands for interchange.

[Iso.org, 2005g] Iso.org (2005g). ISO/IEC 7816-5:2004 Identification cards - Integrated circuit cards Part 5: Registration of application providers.

[Iso.org, 2005h] Iso.org (2005h). ISO/IEC 7816-6:2004 Identification cards - Integrated circuit cards Part 6: Interindustry data elements for interchange.

[Iso.org, 2005i] Iso.org (2005i). ISO/IEC 7816-7:1999 Identification cards - Integrated circuit(s) cards with contacts Part 7: Interindustry commands for Structured Card Query Language (SCQL).

[Iso.org, 2005j] Iso.org (2005j). ISO/IEC 7816-8:2004 Identification cards - Integrated circuit cards Part 8: Commands for security operations.

[Iso.org, 2005k] Iso.org (2005k). ISO/IEC 7816-9:2004 Identification cards - Integrated circuit cards Part 9: Commands for card management.

[Iso.org, 2006] Iso.org (2006). ISO/IEC 7816-3:2006 Identification cards - Integrated circuit cards Part 3: Cards with contacts Electrical interface and transmission protocols.

[Iso.org, 2007] Iso.org (2007). ISO/IEC 7816-2:2007 Identification cards - Integrated circuit cards Part 2: Cards with contacts Dimensions and location of the contacts.

[Karame et al., 2012] Karame, G. O., Androulaki, E., and Capkun, S. (2012). Double-spending fast payments in bitcoin. In *ACM Conference on Computer and Communications Security*, pages 906–917.

[Kocher et al., 1999] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Advances in cryptology, Crypto 1999*, pages 789–789. Springer.

[Kocher, 1996] Kocher, P. C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113.

[Künnemann, 2015] Künnemann, R. (2015). Automated backward analysis of PKCS#11 v2. 20. In *International Conference on Principles of Security and Trust*, pages 219–238.

[Lim et al., 2014] Lim, I.-K., Kim, Y.-H., Lee, J.-G., Lee, J.-P., Nam-Gung, H., and Lee, J.-K. (2014). The analysis and countermeasures on security breach of bitcoin. In *International Conference in Computational Science and Its Applications*, pages 720–732.

[Litke and Stewart, 2014] Litke, P. and Stewart, J. (2014). Cryptocurrency-stealing malware landscape. technical report. *Dell SecureWorks Counter Threat Unit*.

[Ludovic, 2016] Ludovic, R. (2016). PCSC-lite, (accessed october 1, 2016). *pcsclite.alioth.debian.org/*.

[Mangard et al., 2008] Mangard, S., Oswald, E., and Popp, T. (2008). *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media.

[Mostowski and Poll, 2008] Mostowski, W. and Poll, E. (2008). Malicious code on java card smartcards: Attacks and countermeasures. *Smart Card Research and Advanced Applications*, pages 1–16.

[Murdoch et al., 2010a] Murdoch, S. J., Drimer, S., Anderson, R., and Bond, M. (2010a). Chip and pin is broken. In *IEEE Symposium on Security and Privacy*, pages 433–446. IEEE Computer Society.

[Murdoch et al., 2010b] Murdoch, S. J., Drimer, S., Anderson, R. J., and Bond, M. (2010b). Chip and PIN is broken. In *IEEE Symposium on Security and Privacy*, pages 433–446.

[Nakamoto, 2008] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *bitcoin. org/bitcoin. pdf*.

[Nohl et al., 2008] Nohl, K., Evans, D., Starbug, S., and Plötz, H. (2008). Reverse-engineering a cryptographic rfid tag. In *USENIX Security Symposium*, volume 28.

[OASIS, 2015] OASIS (2015). PKCS#11 cryptographic token interface base specification version 2.40.

[Poulsen, 2011] Poulsen, K. (2011). New malware steals your bitcoin. `www.wired.com/2011/06/bitcoin-malware`.

[Quisquater and Samyde, 2001] Quisquater, J.-J. and Samyde, D. (2001). Electromagnetic analysis (ema): Measures and counter-measures for smart cards. *Smart Card Programming and Security*, pages 200–210.

[Raffelt et al., 2005] Raffelt, H., Steffen, B., and Berg, T. (2005). Learnlib: A library for automata learning and experimentation. In *International Workshop on Formal Methods for Industrial Critical Systems*.

[Rosenfeld, 2014] Rosenfeld, M. (2014). Analysis of hashrate-based double spending. *CoRR*, abs/1402.2009.

[RSA Laboratories, 2009] RSA Laboratories (2009). PKCS#11 v2.30: Cryptographic token interface standard.

[RSA Laboratories, 2012] RSA Laboratories (2012). PKCS#5 v 2.1: PBKDF2 Password Based Key Derivation Function 2. *https://www.emc.com/collateral/white-papers/h11302-pkcs5v2-1-password-based-cryptography-standard-wp.pdf.*

[RSA Security INC, 2004] RSA Security INC (2004). v2.20. PKCS#11: Cryptographic Token Interface Standard.

[Ruiter, 2015] Ruiter, J. D. (2015). *Lessons learned in the analysis of the EMV and TLS security protocols*. PhD thesis, Radboud University Nijmegen.

[Saitta et al., 2005] Saitta, P., Larcom, B., and Eddington, M. (2005). Trike v.1 methodology document [draft]. `dymaxion.org/trike/Trike\_v1\_Methodology\_Documentdraft.pdf`.

[Scerri and Stanley-Oakes, 2016] Scerri, G. and Stanley-Oakes, R. (2016). Analysis of key wrapping apis: generic policies, computational security. In *Computer Security Foundations Symposium*, pages 281–295. IEEE.

[Schneier et al., 1999] Schneier, B., Shostack, A., et al. (1999). Breaking up is hard to do: modeling security threats for smart cards. In *USENIX Workshop on Smart Card Technology*.

[Skorobogatov, 2005] Skorobogatov, S. P. (2005). *Semi-invasive attacks: A new approach to hardware security analysis*. PhD thesis, University of Cambridge.

[Steel and Bundy, 2005] Steel, G. and Bundy, A. (2005). Deduction with xor constraints in security api modelling. In *International Conference on Automated Deduction*, pages 322–336.

[Swiderski and Snyder, 2004] Swiderski, F. and Snyder, W. (2004). Threat modeling. *Microsoft Press*.

[The Bitcoin Wiki, 2014a] The Bitcoin Wiki (2014a). *en. bitcoin. it/ wiki*.

[The Bitcoin Wiki, 2014b] The Bitcoin Wiki (2014b). Bitcoin protocol specification. *en. bitcoin. it/ wiki/ Protocol% 5Fdocumentation*.

[The Bitcoin Wiki, 2014c] The Bitcoin Wiki (2014c). Wallet encryption. *en. bitcoin. it/ wiki/ Wallet% 5Fencryption*.

[Tribbleagency.com, 2012] Tribbleagency.com (2012). Bitcoin ewallet vanishes from internet. *www. tribbleagency. com/ ?p= 8133*.

[Turuani et al., 2016] Turuani, M., Voegtlin, T., and Rusinowitch, M. (2016). Automated verification of electrum wallet. In *Financial Cryptography and Data Security International Workshops, BITCOIN, VOTING, and WAHC*, pages 27–42.

[UcedaVelez and Morana, 2015] UcedaVelez, T. and Morana, M. M. (2015). *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*. John Wiley & Sons.

[Uzun et al., 2007] Uzun, E., Karvonen, K., and Asokan, N. (2007). Usability analysis of secure pairing methods. In *International Conference on Financial Cryptography and Data Security*, pages 307–324. Springer.

[Wang et al., 2009] Wang, Z., Jiang, X., Cui, W., Wang, X., and Grace, M. (2009). Reformat: Automatic reverse engineering of encrypted messages. In *European Conference on Research in Computer Security*, pages 200–215.

[Wuille, 2014] Wuille, P. (2014). Dealing with maellability. *Online specification for BIP62*.

[Wuille, 2017] Wuille, P. (2017). Hierarchical deterministic wallets. *Online specification for BIP32*.

[Xu and Nygard, 2006] Xu, D. and Nygard, K. E. (2006). Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265–278.

[Xu et al., 2012] Xu, D., Tu, M., Sanford, M., Thomas, L., Woodraska, D., and Xu, W. (2012). Automated security test generation with formal threat models. *IEEE Transactions on Dependable and Secure Computing*, 9(4):526–540.

[Youn et al., 2005a] Youn, P., Adida, B., Bond, M., Clulow, J., Herzog, J., Lin, A., Rivest, R., and Anderson, R. (2005a). Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge.

[Youn et al., 2005b] Youn, P., Adida, B., Bond, M., Clulow, J., Herzog, J., Lin, A., Rivest, R. L., and Anderson, R. (2005b). Robbing the bank with a theorem prover. Technical report, Univerisity of Cambridge.