Masaryk University
Faculty of Informatics

Distribution Portal For Applications For Cryptographic Smartcards

Bachelor's Thesis

Jiří Horák

Brno, Fall 2020

Masaryk University
Faculty of Informatics

Distribution Portal For Applications For Cryptographic Smartcards

Bachelor's Thesis

Jiří Horák

Brno, Fall 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jiří Horák

Advisor: doc. RNDr. Petr Švenda Ph.D.

Acknowledgements

I would like to thank my advisor, Petr Švenda for guidance and answering countless questions; and my sister, Anežka Wiewiorková for the text review — an author's eye is often blind. I am grateful to Monet+/AHEAD iTec for their interest and support in smart card open-source projects. I would like to express my gratitude to Lukáš Zaoral, Natália Jankaničová and others who helped with the user testing. Last but not least, I wish to thank to my family and friends for their support.

Abstract

Smart card open-source software is difficult to get started with. The software is often poorly documented, uncompiled and the installation can be complex. Moreover, usage requirements and tutorials are usually missing in the software description. JCAppStore introduces easy and secure smart card management, providing automatic security features and compiled software along with use description, examples and links.

Keywords

Smart Card, JavaCard, Deployment, Platform, Store, Applet, Package, Software, Security, Manager, GlobalPlatform, GlobalPlatformPro, Knowledge base, Swing, GitHub, Java, PGP, GnuPG, Keybase, Jubula, GPShell, U2F

Contents

List of Figures

Abbreviations

AES Advanced Encryption Standard; symmetric cipher also known as Rijndael

AID Application Identifier; an unique applet identifier on a card

APDU Application Protocol Data Unit; a protocol used when communicating with a smart card

API Application Programming Interface; a set of means the software offers to use from outside; a set of public classes and methods of a library for example

ASN.1 Abstract Syntax Notation One; cross-platform data serialization standard — data structure definition

AUT Application Under Test; an agent connecting the tested application with ITE

CLA a command APDU part – class; specifies the action type

CLI Command Line Interface; an application interface based on text

DDoS Distributed Denial of Service; an attack on a service by sending many requests at the same time trying to overload its capabilities

DEK Data Encryption Key; used to encrypt sensitive data when communicating with a card

DER Distinguished Encoding Rules; encoding rules for ASN.1 — how the ASN structure should be encoded

ECC Elliptic Curve Cryptography; asymmetric cryptography algorithms based on arithmetic operations with points on an elliptic curve

ENC Encryption Key; key used to derive a key for secure channel establishment

GPPro GlobalPlatformPro; a smart card tool supporting GlobalPlatform-compliant smart cards

GUI Graphical User Interface; an interface providing graphical user interaction

INI Initialization or configuration text file type; widely used

INS a command APDU part – instruction; specifies the action to perform

IP Internet Protocol; used in this document when referring to an IP address — "unique" network node identification

ISD Issuer Security Domain; an SD uploaded by a card issuer

ISO/IEC the International Organization for Standardization and the International Electrotechnical Commission association; manages IT standards

ITE Integrated Testing Environment; an environment designed to handle extensive program testing

JCRE Java Card Runtime Environment; a smart card execution environment (similar to operating system)

JSON JavaScript Object Notation; lightweight data format popular among network applications

JVM Java Virtual Machine; Java interpreter

KCV Key Checksum Value; a value used to validate or compare keys without the actual key value knowledge

LC a command APDU part; specifies the length of command APDU data field

LE a command APDU part; specifies the expected data length in response; missing if not specified

MAC Message Authentication Key; used to derive keys for secure channel establishment

NDEF NFC Data Exchange Format; an NFC message data format specification

NFC Near Field Communication; a set of protocols for short distance wireless communication[1]

P1 a command APDU part; first parameter of an instruction; the meaning is dependent on the applet API

P2 a command APDU part; second parameter of an instruction; the meaning is dependent on the applet API

PC/SC Personal Computer/Smart card; a specification for a smart card integration on computers; part of a Windows platform; available as a free software for other platforms

PGP Pretty Good Privacy; an application program capable of security operations (encryption and signatures); more implementation variants are available

RID Provider Identifier; first 10 characters (5 bytes) of a hexadecimal AID value that identify the software or standards provider

RSA initials of the algorithm founders; popular asymmetric cipher

SCP Secure Channel Protocol; a protocol that is used to establish a secure channel; more variants (e.g. SCP03)

SD Security Domain; an applet that manages the card

SDK Software Development Kit; a software bundle by Oracle; necessary for a smart card development; with backward compatibility; the latest version is 3.0.5

TLS Transport Layer Security; communication security protocol used with networks

TTL Time Ti Live; a counter decreased each time a packet crosses a network node

U2F Universal 2nd Factor; a standard for two factor authentication (2FA) introduced by Google and Yubico

vJCRE Virtual Java Card Runtime Environment; a testing tool by Martin Paljak

XML Extensible Markup Language; widely used markup language

1 Introduction

A smart card, also called JavaCard, is one of the most widely used technologies with respect to number of devices deployed; known as sim cards, debit and credit cards, virtual wallets, etc.

Despite this, the majority of smart card software tools are complete business solutions meant for other companies; too expensive for individuals. Though there are several open-source projects available, the problem is that actual usage of these requires almost a developer knowledge. It is not very convenient either; an example is an applet identifier similar to an IP address. But unlike IP addresses, identifiers are not hidden from the user and a manager expects an identifier to be used when communicating with an applet.

Thus the open-source community is small. The problem is, closed-source security products have a key disadvantage in possible vulnerabilities not discovered in the development process, whereas open-source projects tend to be more secure due to public implementation assessment. Also, IT security features should be open to everyone.

The goal is to create an environment for smart card software deployment, both user-friendly and secure. It includes simplifying management routines, performing user data caching, and growing knowledge database.

We introduce JCAppStore; an applet (smart card software) store that contains hand-picked applets using strict criteria on usability and possible use cases, accessed through a transparent GitHub repository. Furthermore, the binaries are generated deterministically, allowing compiled source origin verification. The store keeps metadata to ease the management and offers knowledge base on applet identifiers and much more.

The second chapter is an introduction to the basics of smart card technology. The third chapter specifies requirements and the fourth chapter focuses on design. The fifth chapter is dedicated to the implementation process, the sixth chapter describes testing and deployment. Finally, the seventh chapter discusses possible extensions and the last chapter is a conclusion. The document further contains an example of applet installation and applet software review for the store.

2 Technology Overview

This chapter is to give the basics of smart card technology. Further chapters assume this knowledge and will not expound such terminology.

2.1 Smart Card Platform Overview

2.1.1 Hardware

The hardware components of a smart card differ based on the card purpose and vendor. However, every card contains following components:

- Processor — central processing unit on which a special version of JVM (Java Virtual Machine) runs
- Co-processor — processor with a special modification for cryptographic algorithms computation; these are too complex for the basic processor to perform in limited environment
- ROM — read-only memory for firmware storage
- RAM — random access memory for computation at runtime
- EEPROM — Electrically Erasable Programmable Read-Only Memory for data storage, with size in tens of kilobytes

All these components are inside a small chip. Other parts may be present, such as antenna for wireless communication, hardware buttons that enforce direct user interaction as another security layer, or even biometric sensors[2] for authentication and more – rather rare – components.

2.1.2 Software API

Every applet has to be developed and compiled using a Software Development Kit (SDK) provided by Oracle. Each card conforms to a specific SDK version. An applet developed with equal (or older)

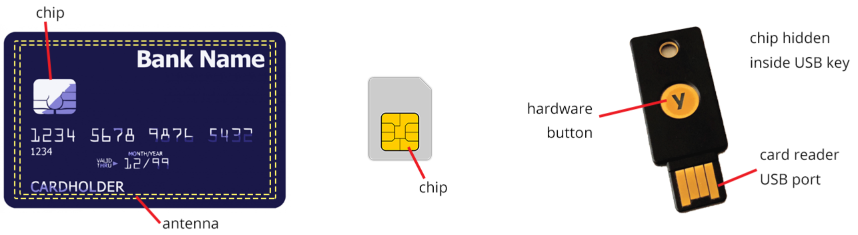


Figure 2.1: Examples of smart cards, sources [3][4][5].

version will work on the card. A large number of cards still conform to SKD version 2.2.2, released in 2006[1].

There are other libraries available, such as commercial JCOP library[2]. However, a card has to support the functionality internally — otherwise, the installation fails with "Conditions of use not satisfied" error message. It mostly concerns various cryptographic algorithm implementations, such as support for elliptic curves. Large database of smart card algorithm support is maintained by my advisor, Petr Švenda[1].

Standard Java and SDK Smart Card API

Supported data types are `byte` and `short` only, from containers an `array`. On the other hand, `javacard.framework` package introduces card-specific functionality.

An applet is a class extending the `Applet` class. This class defines methods called when a specific event occurs. A developer can override them to provide custom functionality. A method `process()` is worth mentioning — called every time the applet receives a command message, a developer defines applet API, the strength of smart card security. For example, a private key generated by an applet cannot be compromised[3], if the applet does not support key-sharing command.

[1] See JCAlgTest database: `https://www.fi.muni.cz/~xsvenda/jcalgtest/table.html`.

[2] Implementation of KeyAgreement.ALG_EC_SVDP_DH_PLAIN_XY algorithm, available since SDK 3.0.5 (see `https://github.com/tsenger/CCU2F`).

[3] Considering software attacks in the applet implementation scope only.

Smart cards do not have the power to run and heavily rely on the readers. To avoid problems with sudden power loss, a collection of `[begin/commit/abort]Transaction()` is available to enclose a part of a code to be run atomically. This is a crucial feature in payment transactions, as either the payment is completed, or not performed at all.

The last characteristic behaviour of an applet execution is error handling. Anytime, an exception can be thrown using specific code, defined in ISO/IEC 7816 standard. If an execution completes successfully, an `0x90 00` code is returned to the reader implicitly.

APDU — Application Protocol Data Unit

APDU is a definition of the communication process between a card and its environment. There are two APDU message types: command (or request) and response. A short rectangle represents one byte (two numbers in hexadecimal, `0xFF` is `1111 1111`) in the following images.

Command APDU

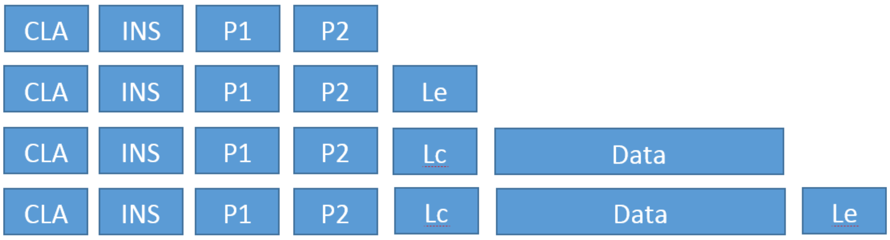


Figure 2.2: Command APDU structure.

The command APDU may be different depending on incoming and outgoing data presence. CLA represents the instruction class of the command. Some values are reserved, as specified by ISO 7816-4. For example, values 0x0_ are used for the SD-specific APDUs (commands to install, modify, delete, manage. . .), values between 0x1_ and 0x7F are reserved, the rest depends on an implementation.

INS specifies the action to perform. An example for CLA of 0x0_ (APDU for Security Domain):

INS	Meaning
CA	Get Data — request information about the card
E6	Install — the actual meaning depends on parameters
E8	Load — uploading the applet onto card

P1 & P2 are instruction parameters. Sometimes, the instruction requires additional information. An example is applet installation. The data array length is limited and applet has to be uploaded part by part. P1 parameter states whether the load is completed, while P2 contains the current block number[6, p. 179]. LC is the incoming data field length. Value of 0x00 is interpreted as a maximum transfer size (usually 256 B). LE represents the expected response data length.

Response APDU

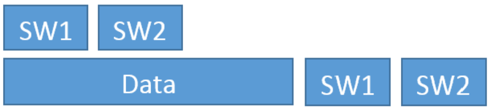


Figure 2.3: Response APDU structure.

The SW1 and SW2 response bytes always carry the result code called status word (SW)[4]. Well-known number 0x90 00 means that the command has been performed correctly.

2.1.3 Applets

Applications running on Java cards are called applets. Applets are usually divided into two categories: ordinary applications and Security Domains (SD). While the ordinary applet is a software uploaded by card user, an SD is usually installed by the card vendor. The SD

[4] The SW consists of two bytes called SW1 and SW2 bytes. The meaning is defined in ISO standard documents ISO/IEC 7816-4, commercial document (https://www.iso.org/obp/ui/#iso:std:iso-iec:7816:-4:ed-3:v1:en). A free brief description is available for example on https://www.eftlab.com/index.php/site-map/knowledge-base/118-apdu-response-list.

performs all the card management, it allows to change the master key or a card state, to install and uninstall applets etc.

Every applet instance belongs to a package. The package represents the context of an applet. Applet instances of the same type (same package ID) share the same context and can easily communicate with each other.

Java Card Runtime Environment (JCRE) separates packages by a firewall. JCRE takes care of the communication between host and the card, between card and the actual applets and also allows to share data between packages via `Shareable` interface.

In the following example, two applets are installed: AP1 and AP2. Application AP1 contains two instances of the program (applets), whereas AP2 has only one instance (the most common scenario).

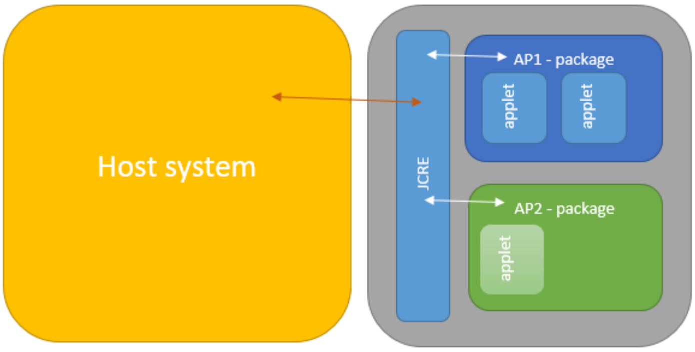


Figure 2.4: Execution environment.

When receiving a command, the SD must decide on who should receive it. By default, the command is sent to the "default selected" applet[5]. The SD can select an applet to temporarily receive future commands using `SELECT` command that carries the applet AID.

2.1.4 Smart Card Life Cycle

As the SD plays this supervisory role over the entire card, its life cycle can be thought of as the life cycle of the card itself[6, p.51].

[5] If not set, the SD is default selected.

2.1.5 Security Domain Life Cycle

1. Operation-Ready or `OP_READY`: the card is ready for management.
2. Initialized: is very similar to `OP_READY`. From GP specification: "The state INITIALIZED is an administrative card production state ... this state may be used to indicate that some initial data has been populated ... but that the card is not yet ready to be issued to the Cardholder."[6, p. 52]
3. Secured: indicates that the card is ready to be issued to its holder (e.g. bank account owner).
4. Locked: indicates that the card management is no longer available. Some applet is chosen as default selected and thus communicated with, and the applet selection may be performed by applets with Final Application privilege only. Applets with Card Lock privilege can swap between Secured and Locked states.
5. Terminated: the SD is no longer usable. Its life cycle has ended. Only the `GET_DATA` command is processed; other commands result in an error.

2.1.6 Authentication

The security domain requires external authentication for some commands to accept. The authentication can be done using both symmetric or asymmetric cryptography, specified by SCP protocol (such as SCP03) [6, p. 138]. SCP defines, for instance, Secure Channel key(s) types[6]. In case of SCP03 protocol, three keys are used for the channel establishment: ENC (encryption key, used to generate S-ENC — session key), DEK (data encryption key, used to encrypt data) and MAC (message authentication key, used to generate session key S-MAC and session key S-RMAC to verify integrity)[7, p. 12]; or it can be one long sequence called Secure Channel Base Key - in this document, we refer

[6] SCP03 is based on AES; other protocols use RSA, ECC or even contactless TLS based approach[6, p. 140].

to it as a Master key. ENC, DEK and MAC keys are derived from the Master key using *diversification method* such as Visa2 or EMV.

2.1.7 Installation

Applet has to be uploaded onto the card first. To do so, `INSTALL` command APDU with parameters indicating the loading process is sent to the card, the data field contains package AID, target domain AID, file hash and load parameters. Next, the file transfer is performed with `LOAD` command. The blocks are sent to the card with a parameter of P2 with the number of the current block transferred. The card loads the data into Executable Load File container and performs integrity check using the given hash once the loading is completed.

Finally, another `INSTALL` command is sent. This time the parameters specify whether and how to install the already uploaded applet. It also carries a data field that is given to a new applet instance (for example a user-defined PIN value).

For a user, the installation process is too complicated. Source code is often uncompiled or provided with poor documentation. The user does not know installation parameters or privileges required, or a smart card master key to authenticate. All data has to be provided in raw form (see appendix A.2). Moreover, an inexperienced user downloads a `.cap`[7] file from the first web-page found and uses it without checking the file integrity — such behaviour defeats the security device purpose. Lastly, the installation process can end up with an error code (`0x68 80` for example) leaving little clue on what might have gone wrong.

2.1.8 Updates

The action of updating an applet with its newer version is very problematic. The hardest part is the applet state preservation (sensitive data). The old applet instance has to be removed first as the AID value must be unique — the case of the same software regardless of the version. Two methods are available: to upload the data into a host system, or to temporarily store the data on a different applet using

[7] Compiled applet binary, similar to Java jar file.

`Shareable`[8] interface. But the purpose of the smart card technology is **hide this data behind the hardware firewall**. Therefore, the data must be encrypted. To establish a virtual secure channel over APDU, following options are available:

1. Update as an applet feature: an applet developer can design an applet and a host application that can mutually authenticate and establish the channel.
2. Update API: The store could define API that applet must implement in order to allow updates; it is not possible unless the store is widely accepted and used.

2.2 Existing Card managers and Applet Databases

A card vendor typically owns a tool for own card management, usually not an open-source. A popular open-source CLI tool is GPShell, implemented as a script interpreter. Another popular command-line tool is GlobalPlatformPro. It is simpler[9] variant of GPShell.

PyAPDU tool is a GUI manager written in Python. But, the main difference is only the GUI layer, more comfortable to use. Ledger Manager is very similar to JCAppStore in terms of interface friendliness and intuitive use. However, Ledger Manager supports its company commercial products only.

While JCAppStore does not offer all the functionality as tools discussed above, the store attempts to simplify the usual card management scenarios as much as possible and verify software signatures automatically. JCAppStore gives the user feedback on what might go wrong when an error occurs. Also, the store is equipped with handpicked applets (see appendix B) including basic use guides for the first steps, links to other useful tutorials and documentation and metadata for better user experience.

[8] For more information see `https://docs.oracle.com/javacard/3.0.5/api/javacard/framework/Shareable.html`.

[9] GPShell requires you to establish context and provide other settings in a script file, whereas GPPro behaves like command-line application with implicit values if not specified. But, GlobalPlatformPro can execute only one action at the time and establish a secure channel each time if necessary.

3 Requirements Specification

"A software requirements specification (SRS) ... describes all the externally observable behaviours and characteristics expected of a software system."[8] The requirements are divided into two categories: functional and non-functional. Functional requirements specify "what" the system should do, while non-functional requirements describe "how" the system should operate. During development, some requirements were discussed and agreed on with my advisor, Petr Švenda. Other arose from user experience testing or applet software reviews.

3.1 Application-specific Functional Requirements

The main purpose of the store is to grow the smart card open-source community. More specifically, the store should:

- enable intuitive applet management,
- manage metadata to improve user-friendliness,
- introduce online central repository for software source,
- offer knowledge database and provide helpful error handling,
- perform software integrity checks.

3.1.1 Store Use Modes

The store shall operate in different use modes. These include:

- verbose mode, for more technical messages and behaviour,
- simple mode, to enable assumptions in actions (e.g. automatically delete package along with the last applet instance),
- exclusive mode, to enable exclusive communication with a card,
- "keep JCMemory" mode, to keep JCMemory applet installed; its purpose is described later.

3.1.2 Card Authentication

The store shall authenticate to a known card automatically. Upon a new card insertion, a default test key should be used. If the card type is unknown, the store shall ask for common test key **0x404142...4E4F** use permission. Successful authentication should be performed automatically henceforth. The user is recommended not to change default test keys (see 3.2.2).

Unsuccessful authentication shall disable further attempts to authenticate. In that case, the user shall provide the correct key value and enable the authentication manually. This will prevent the card from being blocked by many incorrect authentication attempts. Only Master Key authentication will be supported.

Furthermore, a smart card the store was unable to authenticate to will be treated as *unauthorized* card (see below).

3.1.3 Applet Contents Listing

The user should be able to filter the display of applets, (issuer) security domains and packages. Only applet instances shall be displayed by default. This simple feature will give the impression of software uniformity, although one applet consists of two entities in most cases.

The store shall display information about applets using managed metadata or AID/RID database respectively. Metadata will allow to display precise information on author, applet name, version and other features such as optional image icon. Otherwise, an applet is to be identified using its AID:

- RID: The store, for example, does not recognize the author nor the name of a security domain, yet according to the AID prefix of 5 bytes the author can be displayed as "Visa International".
- AID: According to our previous example, the name will be "(VISA) Card Manager" instead of "A000000003000000".
- Other information will stay unknown.

Unknown applet name shall be replaced by its AID, other values described as "unknown". Custom applet installation shall enable to provide this metadata manually.

Unauthorized Card

An user will be able to display some card's installed applets. Also a deep scan feature shall be available, that detects all installed instances. In this mode, any management shall be unavailable except raw APDU message sending. The *packages* checkbox will be hidden as our application is unable to detect packages.

3.1.4 Basic Installation Process

Basic applet installation shall be simple. The user will be able to either install the latest version or select any other combination if available. Before installation, a software signature shall be verified. Also, an available card memory should be assessed to prevent installation failure. The user shall be notified if the installation succeeded and given feedback on possible error cause otherwise. Successful installation should record:

- AID as a primary key,
- applet or package instance name and author,
- applet and SDK version,
- optional image and one flag value indicating sensitive data presence.

Unauthorized card shall not be able to install applets.

3.1.5 Advanced Installation Options

Some applets require additional information for installation. This information should be included in the applet description. The store should fill some values automatically if possible: required default-selected privilege or necessary applet AID modification.

3.1.6 Metadata Management

The store shall record events — such as card authentication — to improve user experience. For card insertion, the following values are to be managed:

1. card name and identifier,
2. card master key, key check value and diversifier,
3. authentication flag for card protection,
4. other information from `GET_DATA` command, to help identify the card manufacturer and product type.

For each installation the applet metadata is to be saved (see Basic Installation Process). Store options file shall be updated with various user personalisation settings that enable:

- using a custom path for dependencies (PGP client may not present in the `$PATH` environment variable or the name differs),
- remembering performance modes,
- latest store content recognition,
- GUI customization and internationalization,
- disabling notifications.

Also, an unauthorized card has no metadata by default. When an user adds some metadata, these will be stored as described above.

3.1.7 Custom Applet Messages

The store should support sending APDU commands. Though this feature decreases the level of intuitiveness, some applets require sending APDU personalisation commands. An user would be forced to download other smart card tools in some cases (see A.1).

3.1.8 Online Central Repository

The store should use central repository with smart card software along with its signatures. The store is to fetch data from this repository automatically.

3.1.9 Knowledge Database Center

Error Handling

Unsuccessful operation should be reported with meaningful message. If verbose mode is enabled, error messages shall contain more technical information along with error codes as received from the card. Otherwise, error messages shall advise on how to proceed assuming inexperienced user: in case the solution should be easy, the possible problem cause will be displayed (such as running out of memory). The error message shall contain only generic failure notice otherwise.

Software Knowledge Base

Unknown software should be identified using its AID. The software provided by the store shall include:

- general information, such as names (possibly more applets in a package), authors, available versions and SDK versions,
- applet introduction and the description of its features,
- use guide that contains basic information, necessary for successful installation and basic usage examples,
- link to the original source code repository and other useful links to documentation and tutorials.

JCAlgTest integration

Card shall be paired with JCAlgTest algorithm test results. These are to be searched for in official JCAlgTest results database. The user shall be able to test their card using our application.

When installing an applet from the store, the algorithm test results are compared to an applet requirements and the user is shown a warning if a card requires some features that were not tested successfully on the card.

3.1.10 Software Integrity Checks

The store should be able to verify software integrity automatically. Furthermore, the store should support the author's signature verification. The store should warn the user in the case of failure.

3.2 Application-specific Non-functional Requirements

The non-functional requirements affect the whole development, thus crucial to consider from the beginning. The sections are written in importance order.

3.2.1 Security

Security is concerned about sensitive data integrity, confidentiality and authenticity. Apart from previously described integrity checks, the store should use transparent strategy. That is, the central software repository should be transparent and the store code open-source so that all claims about the security level can be verified anytime. The store should offer software that was compiled using deterministic build to allow source code origin verification. The store should publish information on how to perform a reproducible build on compiled software and possible changes that could have been made to the applet source code in order to successfully compile.

On the other hand, it is not necessary to encrypt the card metadata, including Master key. The purpose of these is card management, so a potential attacker can only delete applets or block the card, but cannot affect the data itself[1].

3.2.2 Safety

Safety means that no harm is done to the surrounding environment. The store should not render any card inoperable, that is the store

[1] It is possible to create a daemon that could steal these credentials and perform such "denial of service" attack once card is plugged in, and the card metadata encryption is one of the possible extensions for the future. However, because many cards still use default test keys, this attack can be performed nevertheless.

should not block any card with many unsuccessful attempts during authentication. The store should not make any assumptions on what to do in case an active applet deletion is concerned. The store shall identify AID collisions and warn before active applet instance deletion.

3.2.3 Usability, Serviceability and Recoverability

The GUI should be intuitive and designed in a self-explanatory way assuming the technology is confusing for beginners. The error messages should either provide solutions or show general failure notifications.

The store shall be able to run again after fatal error. The metadata kept by the store should be updated only after successful action to preserve integrity. The store should provide an option to refresh its components — meaning the loaded card data and the store contents. Action (such as installation or sending APDU commands) data input should be validated before processing to stop from executing in case the syntax is incorrect. Any user input should be also recovered when action aborted. For example, in the case of invalid installation parameters syntax, the user input should not be lost.

Logging and Crash Reporting

To facilitate debugging and issue reporting, all significant events should be recorded. The store shall log all actions and errors. A logger console and log files should be available for debugging purposes. Furthermore, in case the store crashes a request for report shall be displayed. The user should be able to provide cause description and attach a log file.

3.2.4 Scalability, Reliability and Availability

Scalability in data performance is not very important, as the amount of data processed by the store is rather small. Each `.cap` file has several kilobytes at most[2]. However, the store should support multiple readers and cards. Also, the server should be able to handle the requests for downloads.

[2] Current store data size is 1 MB, including compiled applets, images and detached signatures.

Reliability is very important, no action should cause the store crash and all invalid states should be handled well. An invalid action or error should not render the store unusable. All user input should be validated, all exceptions handled and all operations ran with timeouts.

The central repository should be available all the time so that software can be downloaded. The store should be able to operate offline if necessary. The reason is that some security actions during installation might require offline environment[3].

3.2.5 Code Extensibility, Modularity And Maintainability

Code should be kept clean and well-maintained to ensure simple extensibility. Major components should be divided into separate containers and connected via interfaces. The code should keep good programming principles, such as uniform naming conventions and well-chosen names, avoid code repetitiveness and long blocks in order to increase readability and ease maintainability.

[3] During the development, applets were encountered that require initialization with cryptographic data such as private keys. An example of PGP offline key generation and smart card advantages see `https://lwn.net/Articles/734767/`.

4 JavaCard Store Design

The JCAppStore design is described in hierarchical order, from dependencies and overall architecture through GUI design to individual use cases.

4.1 Development Language and Dependencies

Development language

Java was a natural choice for development. Though there are many advantages such as simple cross-platform deployment or simple, yet powerful multi-threading support, the main reason is that the smart cards — and most of the related software — run on Java.

Graphical User Interface Library

Swing was selected for GUI development. It is a simple and well-documented library. Swing is also lightweight and seemed to be the right choice. Later-showed problem was the GUI components are closely tied with a selected look and feel, difficult to adjust. Because Swing library is rather old, any default design was insufficient and many custom GUI classes had to be created. On the other hand, the support for HTML rendering proved to be useful. Many GUI testing frameworks support Swing too.

Global Platform library

In order to communicate with cards, GlobalPlatformPro[1] was chosen to integrate with the store. This tool is developed with a focus on simplicity through automatic security domain detection and other implicit actions. At first, the tool was one Java package and the integration was done by copying the repository source code, later the structure was improved and the backend part is available now on Maven, included using Gradle build tool.

[1] The author Martin Paljak owns many successful open-source projects including Applet Playground - a project for simple applet compilation, GPPro - popular smart card CLI manager or vJCRE - virtual smart card run-time environment.

Miscellaneous

GnuPG was used to verify software signatures. For logging, log4j library was chosen due to transitive dependency from GlobalPlatformPro. For Windows installation wizard creation, IzPack was used because of rich documentation and powerful features and Launch4j library was used to convert `.jar` files to windows executable (`.exe`). Additionally, the store uses ini4j library for INI file parsing and MigLayout as a versatile GUI layout manager. Testing is performed using Jubula test framework as described in section 6.4.

4.2 Architecture

JCAppStore is a desktop application. All user data is stored locally on the machine. The store content is hosted on a GitHub[9], downloaded for the JCAppStore to work with locally. The following reasons led to the architecture final design:

- Standalone application is minimalist, easy to manage. Local storage does not require accounting, provided the user uses the same computer for card management — the most common scenario with smart cards. But mainly, no server has to be maintained.
- GitHub is reliable and stable. Again, no server maintenance is required.
- GitHub is transparent. The distributed software can be assessed independently of our solution.
- The store data is small in size (1 MB in total for the current release). Moreover, the store works offline once the content is downloaded.

4.3 Application Data and Formats

Beside application source files, the a `JCAppStore` folder is created in default user home directory: `C:\Users\[username]\` on Windows and `~/` on Linux respectively. Following folder hierarchy is created:

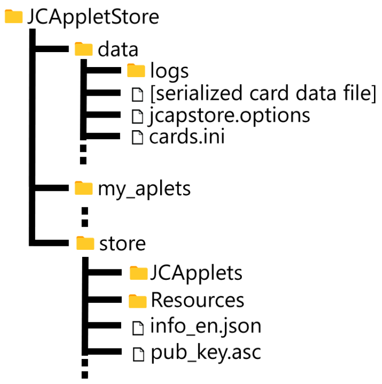


Figure 4.1: JCAppStore folder structure, source[10].

The data folder contains important cards.ini file that stores card authentication data as described in 3.1.6. This file has to be modified to customize a card authentication. This folder also contains results/ folder that stores JCAlgTest algorithm support test result files. The actual *result-to-card* mapping is stored outside the folder in files created as serialized metadata Java class.

The my_applets folder is empty, meant for the user to store custom .cap files. Most of the local data is stored using INI file format that allow simple modification (such as specifying custom Master key). The store folder contains the “JCAppStore Content” repository[9] latest release. The store content uses info_[language].json JSON file when loading.

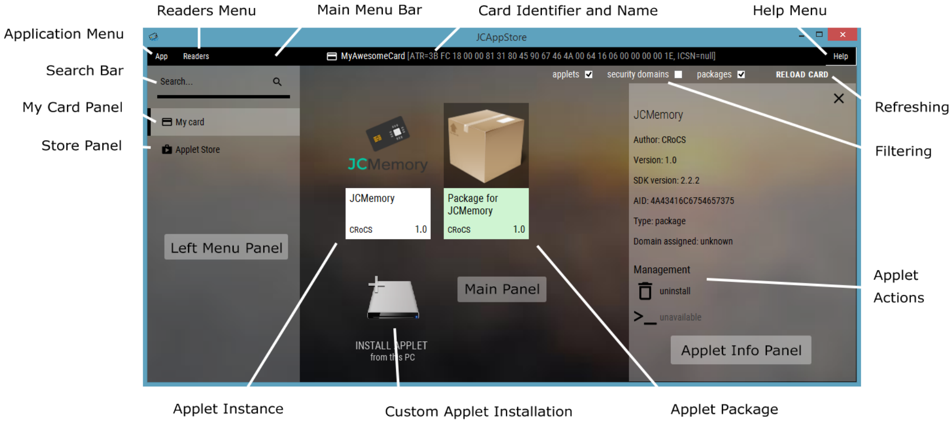
The JSON file can be managed using a CLI application JCAppStoreParser[11], developed in C# course as a final project, written using .NET technology. Therefore, this tool can run only on Windows. It supports signatures generation, translations management, interactive editor and data syntax and semantics validation. requirements.txt files are stored where compiled sources are located. These files list required SDK API calls.

4.4 Localization

When missing a language property, JCAppStore uses default JVM locale if supported. Default language is English. To add a new language, a translation file must be provided in the application resources data folder `src/main/resources/` relative to the `.jre` executable. The file name format is `Lang_[tag].properties` where "tag" represents the language. In `data/jcappstore.options` file, `lang = [tag]` property must be set. To set the property from application, the new language must be added as an `Enum` value.

4.5 GUI Design

The JCAppStore has a main menu bar at the top, left menu container and a large panel that occupy the rest of the window: either "My Card" panel for card management or "Applet Store" panel for browsing the store content. Only one panel is displayed at the time.



MyAwesomeCard [ATR=3B FC 18 00 00 81 31 80 45 90 67 46 4A 00 64 16 06 00 00 00 1E, ICSN=null]

JCMemory
CRoCS 1.0

Package for JCMemory
CRoCS 1.0

JCMemory

Author: CRoCS

Version: 1.0

SDK version: 2.2.2

AID: 4A43416C6754657375

Type: package

Domain assigned: unknown

Figure 4.2: JCAppStore GUI.

4.5.1 Main Menu Bar

In the left corner, the main menu contains two items: `App` and `Readers`. `App` menu item consists of `Card` item that allows the user to obtain available persistent memory on the inserted card or run JCAlgTest process and perform ; `Display` item to show logging console or enable hint pop-ups; `Modes` item that contains available modes as described in 3.1.1; `Settings`[2] and `Quit` item to exit the application.

`Readers` menu item enables switching between available smart card readers. On a new smart card reader detection, the item notifies the user by temporary colour change. The midsection of the menu bar is occupied by currently active card name and identifier. The icon next to the card name identifies the card status as follows:

1. crossed card icon that indicates no card has been detected,
2. normal card icon that means the card has been authenticated to,
3. card with lock icon that warns an user the card authentication failed and majority of management features are not available,
4. card with question mark icon that reminds user this card has no JCAlgTest data paired.

The `Help` menu item in the right corner offers FAQ.

4.5.2 Left Menu

The Left Menu panel allows switching between two Main panels. The Search Bar filtering applies to the currently displayed Main panel. A search query can be any substring in the applet name, author or applet category (if searching in the applet store).

4.5.3 Applet Store Panel

Store loading and applet listing

In case no internet connection is available, the latest local store copy is displayed. Otherwise, the latest release[9] is downloaded if newer version found. The applets are displayed by categories (payment, authentication, utilities ...) and the display order is defined by JSON info file. The store window has another submenu at the top that allows "go back" action and manual re-downloading of the whole repository.

[2] Contains language selection, custom PGP installation path and loading custom background image.

Store item details

Selecting an item opens detailed info window. The user is shown a brief applet description, installation and basic use guidelines, a set of related websites and a link to the applet official repository. Furthermore, two "INSTALL" buttons are displayed - one at the top that installs the latest version of the latest SDK version. Another button is at the bottom, allowing a custom version selection.

4.5.4 'My Card' Panel

Once a smart card insertion is detected, the store automatically authenticates to the card if no other card is being displayed (for the authentication process, see 4.6.3). Once authenticated, the card panel displays installed entities if possible. An error message is shown in case the authentication failed or another error has occurred:

- "No card service enabled.", in case the smart card service is not running;
- "No card reader found.", if no smart card reader detected;
- "No card in the selected reader.", if the connected reader is empty;
- or a message describing the reason behind the failure.

The installed entities are shown as described in the requirements section 3.1.3. Apart from filtering, `RELOAD CARD` button is available to refresh the card contents manually.

If the card was authenticated to, another button is present on a single last row. This button allows custom applet installation. Selecting it opens File Chooser to locate a `.cap` file. Selecting any other item opens applet info panel with information details. The info panel also provides two actions — entity deletion and sending a raw APDU command. The button title says "unavailable" if the action is invalid for the instance type.

4.5.5 Logging Console

By default hidden, a logging console is available in the store. The console is useful for an experienced user to quickly get more technical details on the most recent failures. Full logs are stored in `data/logs/` folder as shown in figure 4.1.

4.6 Important Use Cases and Backend Design

This section contains a description of the most important use cases and backend behaviour.

4.6.1 GlobalPlatformPro Wrapper: Card Manager

GlobalPlatformPro is able to handle all the functionality when communicating with a card. However, another abstraction layer is needed in order to add more features and hide the library low-end API behind interfaces. Metadata management is performed: card detection, key guessing, applet data serialization, RID- and AID-based software recognition, card reader "multitasking", logging and multithreading.

4.6.2 Multiple Readers Behaviour

If the currently selected reader does not have a card inserted, another reader with a card is selected instead. Otherwise, the selection remains intact.

4.6.3 Authentication

This activity diagram describes the process of authentication. The user can customize this process for each card providing KCV, diversification method and Master Key value in the `cards.ini` file.

4.6.4 Applet Installation From the Store

At first, the software signature verification is performed and installation context window shown along with the signature result. This window contains applet information discussed in section 3.1.4, and "Advanced options" button. After the window confirmation, the store

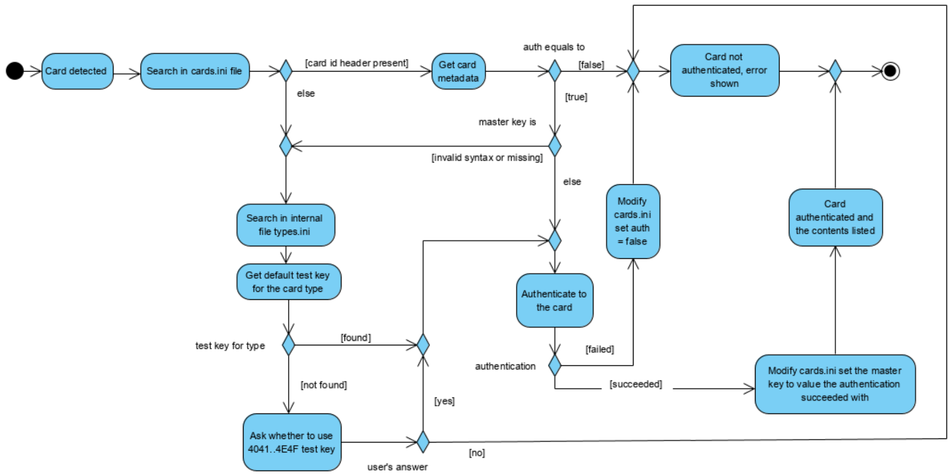

Card detected
Search in cards.ini file
[card id header present]
else
Get card metadata
auth equals to
[false]
[true]
master key is
[invalid syntax or missing]
else
Card not authenticated, error shown
Modify cards.ini set auth = false
Card authenticated and the contents listed
Search in internal file types.ini
Get default test key for the card type
test key for type
[found]
[not found]
Authenticate to the card
[failed]
authentication
[succeeded]
Modify cards.ini set the master key to value the authentication succeeded with
[yes]
Ask whether to use 4041..4E4F test key
user's answer
[no]

Figure 4.3: Authentication process activity diagram.

measures available card memory and a warning is displayed in case the memory size can be insufficient. Otherwise, the package is loaded and requested applet instances created. The applet may require default selected privilege for use, in this case, the privilege is added to the corresponding applet instance automatically when created. Lastly, the user is notified whether installation succeeded.

4.6.5 Simple and Advanced Installation

Some applets require additional information when being installed. Simple successful installation supposes that only one applet instance is created (the first one if multiple present), no installation parameters are required and no collision in package or applet AID occurs.

In other scenarios, the advanced section has to be adjusted. It is hidden in the beginning so that the installation context window looks simple. Once visible, the advanced section shows a combo box with applet instances, names AID values enabled for modification. Applets to install can be selected there[3].

Installation parameters field offers an option to pass data to the applet when installing. Some applets require personalisation data for

[3] Screenshots are available in appendix A.3. No applet is also a valid choice, in that case only the package is created on the card.

example. This data must be in a hexadecimal form. The installation parameters structure is defined by applet developer and thus impossible to simplify. Therefore, both applet AID and parameter fields include input sanity checker.

The user can also select the “force” option — that is, any AID collisions are removed before the installation proceeds. A warning is displayed in this case by default.

4.6.6 Applet Installation Process Diagrams

The most accurate description is provided in following activity diagrams.

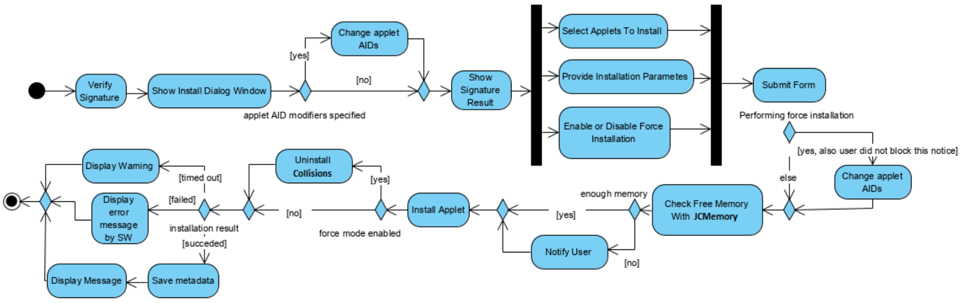


Figure 4.4: Installation from the store activity diagram.

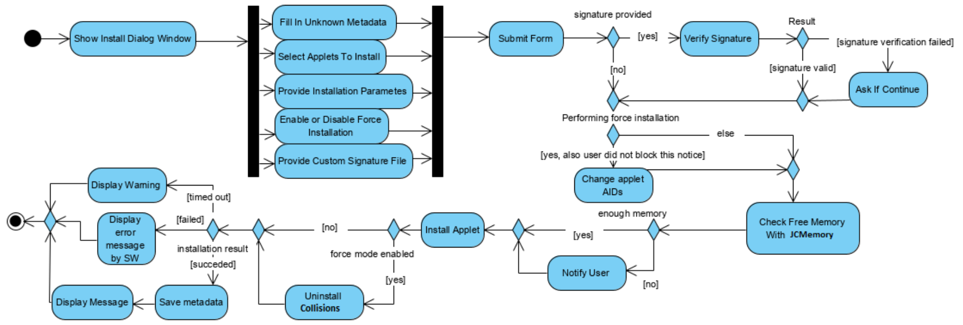


Figure 4.5: Custom applet installation activity diagram.

4.6.7 Instance Deletion

In case the deletion subject is marked as an applet storing personal data (security keys, passwords etc.) the user is asked about the deletion twice: a deletion dialog window and data loss warning. If the deleted instance is a package and force deletion is enabled, "simple mode" is similar to uninstalling with force option enabled: all applet instances must be removed too. The user is notified in case any applets are found on the card that would be uninstalled and the process must be confirmed.

If uninstalling an applet instance, "simple mode" removes package if the applet is the only one instance of that package. Force deletion is not allowed in this case.

4.6.8 Downloading the Store Contents

At first, the store tries to connect to GitHub and download a JSON file that describes the store content repository[9] latest release. If the saved release name differs or the store folder is missing crucial files, the store downloads the newest release, unpacks it and loads the content.

4.6.9 Sending Custom APDU Messages

Installed applet instances offer sending APDU on the info panel. Similar to installation parameters, the user has to provide a hexadecimal command that is sent to the applet. The applet selection is done automatically.

4.7 Crash Reporter

JCAppstore includes a crash reporting feature — in case fatal error occurs and the client crashes, a reporting window is shown. The user is provided both with an error message that describes the failure and a log file. A link to our repository issues webpage is displayed with information on how create an issue report. The JCAppStore repository[12] contains predefined templates that help creating an issue.

5 Implementation Phase

The JCAppStore description was fully covered in previous chapters. The following text describes selected implementation parts and problems the development had to deal with.

5.1 Clean Code

The component dependency realization is always problematic. The GUI class structure is hierarchical, having as a root component the `JCAppStore.java` main application class serving as a JFrame (window) container. Other JPanel-extending classes or components are included in the frame (the Applet Store and the "My Card" section, for example) and these include other sub-menus or panels. Many components are dependent on others and passing everything in constructors would quickly pollute the code.

Another problem is code repetitiveness, the store had to be refactored in the middle of the development process to remove duplicate code and increase maintainability.

5.1.1 Packages

Consisting from more than 150 classes, the store is separated into several packages:

- `.appletstore`: contains the major GUI classes,
- `.action`: bridging actions between GUI and backed, these implement installations, cards detection and other functionality,
- `.card`: card manager implementation, contain direct communication with underlying smart card libraries,
- `.card.command`: classes that extend the CardCommand interface, these can be passed to a `CardInstance` class to perform,
- `.crypto`: signatures verification implementation,
- `.help`: classes that display help panels,

- .ui: various Swing extensions to define custom design,
- .util: a collection of parsers, loggers, asynchronous workers and other utility classes.

5.1.2 Logical Groups and the Factory Pattern

The store makes use of several enclosed logical units that are represented by interfaces to increase code modularity. Major units are:

- Card Manager — a backend smart card manager,
- Options — a store personalisation data handler,
- Informer — a component that removes cyclic dependencies and solves asynchronous message handling,
- Signature — representing the crypto package, the Signature interface can verify .cap file signatures,
- CardManager & CardInstance — the major components the smart card are used through in action package,
- CardCommand — interface required by CardInstance when processing a command, used inside card manager only.

These are needed in different parts of the code. The obvious solution is a Singleton "anti-pattern"[1].

```
class Singleton:
      private static instance
      public static function get():
          if instance is null then:
              instance = new Singleton()
          return instance
```

The problem is, a singleton introduces a global variable to the system. The components become dependent on the singleton and the code becomes harder to test, maintain and re-use.

[1] It is an overused pattern for its simplicity, thus often referred to as an "anti-pattern" instead of a "pattern". The "anti" serves as a good reminder that one should think twice before using it.

Though using a singleton would not be wrong in this case as only GUI-related classes need these, a Factory pattern is more suitable. The management of the singleton instance as shown before can be separated into a different class named `[Singleton Name]Factory`. Unlike `Singleton` itself, a `Factory` class is lightweight, can return different instances (of the same implementation) and mainly, preserve dependency inversion principle. A singleton conforms to an interface that cannot be instantiated. Thus a singleton implementation is partially exposed to the outside, whereas factory pattern dependency is only abstraction-based.

Factories also add another layer between singleton and its user: an abstract factory can be instantiated with different implementations, providing a singleton implementation change without touching any other code[2]. And lastly, the Singleton creation is consistent, not related to the Singleton implementation[13].

Cyclic dependencies

Reporting an action result creates a cyclic dependency in the case the message is to be displayed on the parent component (not a pop-up window). The `Informer` interface is a component that has to be initialized with an `Informable` class. Once initialized, any other class can display a message using the `Informer`. These messages can come asynchronously, include a callback operation to perform or auto-close after a given time. For this purpose, a `LinkedBlockingQueue` was used. Once the first message is received, a new thread is created and the message is pushed into the queue. This thread requests the `Informable` object to display the message at the top of the queue and call

```
Thread.sleep(message.timeout)
```

afterwards. The procedure is repeated until the queue is empty.

5.1.3 Abstract Wrappers

A wrapper is understood here as a container implementing common functionality and thus reducing code repetitiveness. An abstract class was usually created to unite common behaviour.

[2] Allowing simple testing using mock implementations.

Card Commands

Different APDU commands have a lot in common. At first, a basic channel must be established (in exclusive mode optionally, see 3.1.1). Once created, APDU commands can be issued. If these commands require a secure channel, it must be created above the currently active channel using a Master Key. Afterwards, the basic channel must be disconnected from. Also, all these steps have to be performed in separate `try-catch` blocks to help identify the possible error cause. It is around 90 lines of code, crucial to perform on a single place to ensure errorless implementation.

`CardInstance` contains two methods called `executeCommands()` and `secureExecuteCommands()`. Both take a variable number of arguments of `GPCommand` type. The first method can establish basic channel whereas the second method is implemented using the first one, passing the secure channel establishment as an anonymous `GPCommand` implementation. Moreover, variable number of arguments allow executing multiple operations with the same channel established, unlike in GlobalPlatformPro tool. For example, installing an applet might consist of any AID collision deletion, desired applet installation and card content listing. GlobalPlatformPro would establish a secure channel three times in this scenario.

Actions

Actions were introduced during the refactoring process. At first, the installation and the deletion processes were simple and implemented as anonymous classes. For instance, there seemed to be no reason for merging installation process from the store and from the custom source, as the processes differ. But soon the code became repetitive in all the card management actions too:

1. Different thread must be used as these actions can last up to tens of seconds.

2. Exceptions may arise that are handled in a similar way.

3. The execution may require a timeout.

4. GUI must be notified and user interaction disabled during the process and enabled afterwards.
5. Invalid or missing Master key error must be handled when establishing a secure channel.
6. Logging must be performed.

The action classes were factored out into a separate package, increasing the abstraction level in GUI classes. Another abstract class was introduced to implement the common behaviour (1-6) and offer a protected method which is executed on another thread, where exceptions are handled, process logged and which can set its own timeout. Though now it seems to be an obvious solution, most of these features were added gradually and the correct approach was not obvious from the beginning.

In the end, the action family grew to contain seven different actions in contrast with the original three. Adding a custom feature or a new action became much easier. Even non-action job — card detection routine — was moved here due to its operational requirements analogy.

The `InstallAction` class is worth mentioning. Thanks to creating a separate file implementation, changes in the action process were simple to make and test. The changes include the free memory verification, signature process changes (see 5.7), policy (at first, implicit applet conflict deletion was implemented) and the installation applet choice from exactly one to any.

5.2 Card Detection Routine

Smart card managers are usually synchronous - that is, the card can handle one request at the time and supporting asynchronous card management is not worth the effort. The user interaction is also disabled when communicating with the card to prevent the user from intervening with the process. Thus the backend implementation could assume synchronous requests and do not implement a mutex[3].

[3] Mutual exclusion mechanism.

But, the store periodically checks for a new reader or a card presence. A terminal list is obtained from a `TerminalManager` every two seconds. The list is compared to cached values. The "Readers" menu item is updated, and if no smart card is being currently displayed and a smart card is found, the card is selected and authenticated to. The implementation of this feature enforced the Card Manager to be thread-safe.

5.3 Detecting Installed Applets On An Unauthorized Card

A `SELECT` command accepts any AID prefix. That means a smart card returns `0x90 00` when any applet instance with AID that match to this prefix is present. The `well_known_aids.ini` file is scanned and a *trie* tree data structure is build from the AID list. Then we walk the tree depth-first, running a `SELECT` command periodically on AID prefixes, cutting branches that do not return `0x90 00`. The period of `SELECT` command call as long as minimum prefix length are customizable.

5.3.1 Deep scan

With the same approach as above, the card is scanned using bruteforce method: instead of having a trie, we simulate the tree by running between `0x00` and `0xFF` in each depth[4] (or prefix length) as long as `0x90 00` is returned. The results are saved as a card metadata by serializing the AID list object.

5.4 Thread-Safe Card Manager

Performing a card action when another was not yet finished would result in shared channel violation. To prevent the card detection routine from intervening with another action, a synchronization is used. In Java, basic synchronization can be done using either `volatile` key-

[4] Tree depth is a number, it is also a layer/level of all nodes with the same length (the number) of their shortest path to the root node.

word[5] to mark an attribute as atomic (a counter, for example), or synchronized keyword to synchronize blocks of code:

```
private volatile Type value;                  //(1)
...
public void callAsynchronously() {            //(2)
    //action
    synchronized(lock) {
       //critical action in relation to lock
    }
    //action
}
...
public synchronized void callSynchronously() { //(3)
    //critical action related in relation to function
}
```

The first example makes an access to the "value" variable atomic, whereas synchronized blocks are performed synchronously in relation to either given object (second example) or a method (third example) — only one thread is allowed to enter at the time (other threads are suspended[15]). In our case, any manager instance must access the same reader synchronously. Assuming the most common scenario of one card reader inserted at time, the second example with the "lock" as a static object is used.

5.5 Smart Card Memory Analysis

To prevent a low-memory installation failure, JCMemory applet is installed on the card. The applet can perform only one action: to return the amount of free persistent memory[6]. The .cap file size (+ 1kB, an approximate applet instance size) is compared to the value returned. In case the available memory is smaller, the user is notified that the installation process is likely to fail.

[5] Unnecessary for most primitive variables, except long and double. Memory inconsistency errors are still possible[14].

[6] Only with the most recent SDK version it is possible to always get the exact value. Older versions use signed short variable. Therefore, the maximum is 32 767 bytes, meaning the card has possibly more free space available.

5.6 Testing the SDK API Support

For each applet, the source code has been scanned[7] for SDK API keywords and the found results has been saved to `requirements.txt` file. Before the actual installation, the card test results are compared to this file and a warning may be shown. The installation can be performed regardless of this action outcome.

The API support test might be inaccurate. Despite removing all comments from the source code before processing, an applet might use other libraries and APIs that we can not test the support for. Also some API calls might not be used in the compiled version because of compiler optimization.

5.7 PGP: Keybase Instead of GnuPG

We attempted to integrate a Keybase client instead of standard GnuPG implementation. Keybase supports most PGP functionality. But unlike PGP, the trust with Keybase is based on history, a network of connections and associated accounts. Even though the PGP signature is valid, the user has to decide for himself whether the public key is trustful. Maliciously modified software signed with a different key can be also successfully verified and therefore the public key origin must be known.

The attempt to integrate Keybase was unsuccessful. Its use as a third-party tool is limited[8], meant for direct use.

Therefore, GnuPG was used instead. Our key origin is not problematic as we add the store's public key to GnuPG keyring and set ultimate trust to it if allowed when installing. But it is still a problem in case an author's signature is included, e.g. signed release by a developer. In this case, the user has to import the developer's public key himself and decide on the trust level. Though this feature is supported, it is not used yet and unlike with Keybase, it is not fully automated.

[7] using JCAppStore Content Parser

[8] Calling the client app through command-line started Keybase daemon. Also, Keybase does not support full PGP features, such as smart card integration as discussed in GitHub issues: `https://github.com/keybase/client/issues/8609`. This may or may not be true in the future as Keybase still evolves.

5.8 Reporting a Crash Event

At first, the Crash Reporter was implemented to work automatically. A window opened for the user with error cause description and option to fill in more details. The user could provide his email address and allow or disallow a log file attachment. This data was sent to Aisa[9] server using `HTTP POST` request.

However, implementing such feature to be secure is problematic. The feature might be misused by attacker to send data periodically to perform a DDoS attack and the defence filter would have to

- compute Hop Count check (subtracting initial and final packet TTL to discover IP spoofing),
- monitor packet frequency (blocking repeatedly used IP address),
- send a logical puzzle in case the IP address is suspicious (at least one of the above apply, CAPTCHA works on similar principle)
- and also verify a signature(s) included in the message header (probably not applicable in our case).

[16] In short, it is always a good idea to keep a KISS[10] principle in mind. Though the intention was to simplify the error reporting as much as possible, the implementation was changed to advise on raising an issue on JCAppStore GitHub repository[12]. The user is provided with a direct link and a button to open the folder with log files.

[9] A student server run by Masaryk University in Brno.

[10] Keep It Simple, Stupid! principle: `https://en.wikipedia.org/wiki/KISS_principle`.

6 Testing, Profiling and Deployment

The JCAppStore is a mainly GUI-based project. Focusing on simplicity through GUI design, there is little to test with standard unit or integration testing methods. Instead, manual testing, user experience testing and automated GUI testing approach were taken as many things described below cannot be achieved with standard testing methods. The aim was to

1. polish error messages and ensure meaningful information is given,
2. adjust GUI design to increase intuitiveness,
3. remove annoying features (e.g. pointless pop-ups),
4. test a user ability to perform basic tasks without a guidance,
5. test whether the application behaves as expected.

In following sections, described aims are to be referenced featuring the list above.

6.1 Manual testing

Manual testing was performed during the final phase of populating the store with software. When compiling and installing over 70 applet software bundles, all of the above except point 4. were adjusted many times. Some examples include:

- pop-up notification about signature result was modified to show only in case the verification fails, an option "don't show this again" was introduced for other warnings,
- store navigation was adjusted to a browser-like behaviour,
- mode for exclusive card communication was introduced, as other simultaneous clients may interfere (Yubico Authenticator in our case),

- available memory automatic test feature was implemented for more robust installation process,
- installation window modality, timeouts, default-selected privilege, multiple applet instance selection and automatic applet AID corrections[1] were added,
- over 90% error messages were continuously adjusted.

6.2 User Experience Tests

These tests were mainly focused on points 2., 3. and 4. A user was given a task to complete — provided with card and installed JCAppStore or an installer, the user was supposed to install and use given applet[2]. Over five people participated in the initial testing. More testing was planned, but was not possible due to COVID-19 related country lockdown.

Nevertheless, the testing did prove useful. The obvious problem with the GUI design was its intuitiveness. People who contributed in this part found the store confusing. The "Applet store" and "My card" sections were almost identical — prone to mislead. The searching bar was clumsy and required confirmation, installed applet instance info panel did not show "close" icon[3] and important components and sections were not highlighted enough[4]. The store was also missing categories, the application settings were too complex, "Readers" menu item did not notify the user upon new card detection and the current card identifier[5] was invisible most of the time. Some explanations

[1] Applet AID can be different from the AID it should be installed with. Consider a package with two applets. The developer has a RID assigned and the package prefix is given. However, one applet is an NDEF tag and its AID is defined (and different) so that host applications find the applet. This NDEF applet has to have AID prefix equal to it's package prefix, but NDEF tags has different RID. JCAppStore can automatically replace these values when installing.

[2] For example, to load PGP applet onto a card, generate RSA keys on the computer and destructively load the private part onto the card.

[3] To close, the user was supposed to click again on the applet instance.

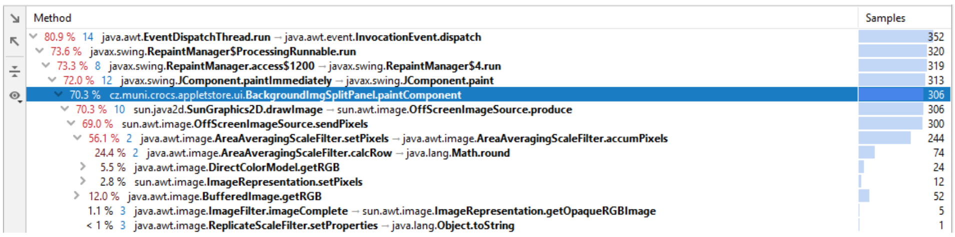
[4] Install buttons, a crucial "Applet use" section in the applet store info screen, missing bold formatting to highlight important information in error messages etc.

[5] The card identifier is necessary for modifying the card database files.

were clarified or added — the most crucial missing information was, the store itself does not offer an environment for the applet usage and other tools are necessary to download. The testing also led to the "Help" menu creation. One performance bug was discovered — transparency issue described in the following section.

6.3 Profiling

We noticed the application ate up to 30% CPU when running, actively used or not during the first user experience testing. The application was adjusting background image every `paint()` call and the image had to be cached. Because every Swing component is responsible for its area, overridden `paintComponent()` method must always redraw the whole rectangle the component occupies if opaque. Problematic were semi-transparent elements. Suddenly, the background was not being continuously repainted and GUI change imprints were being stacked[17].



```
Method                                                                                                      Samples
∨ 80.9 % 14  java.awt.EventDispatchThread.run → java.awt.event.InvocationEvent.dispatch                      352
 ∨ 73.6 %  javax.swing.RepaintManager$ProcessingRunnable.run                                                 320
  ∨ 73.3 % 8  javax.swing.RepaintManager.access$1200 → javax.swing.RepaintManager$4.run                      319
   ∨ 72.0 % 12  javax.swing.JComponent.paintImmediately → javax.swing.JComponent.paint                       313
    ∨ 70.3 %  cz.muni.crocs.appletstore.ui.BackgroundImgSplitPanel.paintComponent                            306
     ∨ 70.3 % 10  sun.java2d.SunGraphics2D.drawImage → sun.awt.image.OffScreenImageSource.produce            306
      ∨ 69.0 %  sun.awt.image.OffScreenImageSource.sendPixels                                                300
       ∨ 56.1 % 2  java.awt.image.AreaAveragingScaleFilter.setPixels → java.awt.image.AreaAveragingScaleFilter.accumPixels   244
           24.4 % 2  java.awt.image.AreaAveragingScaleFilter.calcRow → java.lang.Math.round                   74
         >  5.5 %  java.awt.image.DirectColorModel.getRGB                                                     24
         >  2.8 %  sun.awt.image.ImageRepresentation.setPixels                                                12
       > 12.0 %  java.awt.image.BufferedImage.getRGB                                                          52
         1.1 % 3  java.awt.image.ImageFilter.imageComplete → sun.awt.image.ImageRepresentation.getOpaqueRGBImage   5
        < 1 % 3  java.awt.image.ReplicateScaleFilter.setProperties → java.lang.Object.toString                 1
```

Figure 6.1: Profiler output in IntelliJ IDEA.

The cause of the problem was these elements *did not* override the `paintComponent()` method. These are essentially opaque elements with semi-transparent background colour, and Swing did not consider it necessary to repaint an opaque element. The solution was to enforce repainting of the semi-transparent background and draw its children items atop.

```
g.setColor( getBackground() );
g.fillRect( 0, 0, getWidth(), getHeight() );
super.paintComponent( g );
```

6.4 Automated GUI Testing

NOTE: the tests are valid for JCAppStore version of 1.0 and have not been modified yet for never versions that bring more features. Further user experience testing was impossible and we focused more on GUI tests using automation tools. Swing proved to be a good choice in this case since many GUI testing frameworks support this library.

Jubula[6]: open-source, powerful Integrated Testing Environment (ITE), was selected. Jubula does not require code modification — it treats the software as a black-box, thus closer to a user perspective. "Jubula tests incorporate best practices from software development to ensure long-term maintainability of the automated tests."[18]. The framework is functional-based — the ITE does not even support traditional copy-paste operation on test cases/parts. This way, a tester is forced to write smaller test parts with generic input parameters and reuse them.

The main focus was on point 5. though other aims were improved as well; file chooser last location caching was implemented for example.

6.4.1 Virtual Cards and CI Servers

Unfortunately, the testing, although automated cannot be run without an assistance before launching. The cards themselves are problematic. There are two components required: a virtual card reader simulating PC/SC interface and a GlobalPlatform-compliant virtual card. The only one open-source virtual reader we have found is the "UMDF Driver for a Virtual Smart Card Reader" by Fabio Ottavi. The driver is difficult to install because it is not signed. The tests were written on a Windows platform and the driver installation required us to temporarily disable compulsory driver signature verification when booting the system. Only after the driver was running we failed to finish this goal due to a simulator.

Although JCardSim is popular and partially-free smart card simulator, GlobalPlatform-compliant version is commercial[7]. Therefore, the

[6] More details on `https://www.eclipse.org/jubula/`.

[7] Approximately one thousand euro a year.

testing requires real smart card device and running on a Continuous Integration (CI) server is impossible.

6.4.2 Jubula Limitations

Although powerful, the tool has its limits. In the following sections, some challenges are described we had to overcome.

Negative testing

Jubula does not support "negative test confirmation". For example, reuse a test case "applet is present on a card" for its negative version. It is not easy to test absence on a component that is missing[8]. Fortunately, this can be achieved using a simple `while-do` sequence:

```
while (test case that should fail) do
  error
done
```

Simply, Jubula can use a `while` condition to accept the "inverted test". We tried to use event handlers[9], but these expect the handler to either successfully complete when repeated, or skip the rest of the test section.

System Dialog Windows

Jubula cannot handle system dialog windows well; the file choosers were very difficult to deal with. The only way to communicate with the application in this situation is to use the keyboard only. Therefore, more strict restrictions are defined under which the test must be run. For example, a custom `.cap` file installation test case requires a specific file to be present in a defined folder. The test case then uses "copy-to-clipboard" and "past" actions to select the file.

[8] Jubula requires the test parts to have all references assigned. This would require us to create a scenario with non-existing references to components. On the other hand, this particular example can be solved also by using "search" feature and verifying that no items has been found.

[9] A test case can be specified to launch on a specific error event.

Test Case Independence

Correct tests start with given state and input and the result do not influence other tests. It is the most difficult feature to achieve. JCAppStore keeps track of many settings to ease the usage; some test suites must rely on other[10]. Furthermore, one test failure usually influences other tests. It is common in GUI testing and Jubula handle this problem the easiest way - a test failure terminates the test execution as a whole.

To an extent, custom behaviour can be defined using event handlers to handle these issues, but the event handler can be defined for the whole test case only.

6.4.3 Preconditions for Test Suite Execution

Due to the reasons discussed above, the following preconditions must be met before the test suite is executed. The following section also describes some conditions common to all Jubula tests, such as the Application Under Test (AUT)[11] setup.

System Environment

GnuPG is installed on the computer, its binary executable accessible via `$PATH` environment variable or specified in the Settings. JCAppStore's public key is imported into PGP keyring. The Internet connection must be available.

Jubula Setup

The test project require AUT to be set up with tested jar executable provided. The AUT must be running and the tested application started. The application should be directly behind Jubula in system window hierarchy — once the test is started, the ITE is minimized and the application should get the focus.

[10] For example, testing the ability to uninstall an applet, the applet has to be present. Thus either this test case has to follow an "install" test case or the "install" case must remove its installed applets and vice versa — e.g. perform install/uninstall twice.

[11] An agent running on user-defined port behaving as a mediator between the ITE and tested application.

JCAppstore Setup

The store setup should be default (as if the store was just installed) and also:

- a smart card must be inserted and no other application must not intervene with it,
- the inserted card must be in `OP_READY` state, fully functional, empty, support at least SDK version of 2.2.2, and the store must be able to authenticate to the card automatically[12],
- the JCAppStore language must be set to "English",
- the `JCAlgTest_v1.7.4_sdk2.2.2.cap` file must be present in the `my_card/` folder, along with `[the cap filename].sig` file for custom signature verification — both files can be copied from the `store/` folder.

6.5 Deployment

Because JCAppStore should be simple to use, we provide following means of installing the store both for Windows and Unix-like systems.

6.5.1 Windows

IzPack is powerful library that allows an installation wizard creation using XML. The installation wizard will ask the user to allow the JCAppStore's public key import.

6.5.2 UNIX

For Unix-like OS, we provide a compiled source and necessary files along with advice on how to proceed:

- all the files except the launcher script should be placed in `/usr/share/java/jcappstore`,

[12] Otherwise, the user is prompted for "4041..4E4F" test key usage confirmation.

- the launcher script shall be modified: the path selected in previous step must be provided,
- the script should be moved to `/usr/bin`, made executable and renamed to any suitable name.

Unlike with Windows, all other installation logic is performed in the launcher script.

Ubuntu

Because Ubuntu is a popular Linux distribution, we provide a debian package (`.deb`) that essentially performs the described procedure automatically.

Lancher Script Description

This script verifies the `GnuPG` existence by calling `gpg --version`. Note that the GnuPG's name should be `gpg` not `gpg2`. Otherwise, either the script shall be modified or the `gpg=gpg2` alias set. The script asks the user to allow the public key import and trust upgrade. Then, an empty file is created to signalize this procedure has been performed and the application is run.

Furthermore, if installed from the package the launcher script changes ownership of the sources directory. Because an installation is run under the root the sources belong to the superuser. The launcher script has to run the `jar` package and to do so, a superuser is required. The ownership cannot be changed during the installation process, because there is no guaranteed way to find out the username the ownership should be changed to. Therefore, the script must be executed using `sudo` for the first time, the username is retrieved from `$SUDO_USER` variable and the ownership changed. After the ownership change, the script exits successfully and requests the user to run it once again under normal privileges.

7 Possible Extensions

Further development is tightly connected to JCAppStore's audience. In case the deployment is successful and the store is frequently used, the following features might be introduced, provided a server is to be maintained:

- API: the store could define API the applets would implement to allow automatic data deduction (SDK required, for example) or advanced behaviour (updates).

- Online database: similarly to Google Play or Apple App Store, the database could be accessible both using a web browser and client application.

- Smartphones support: the popularity of these devices grows rapidly. For example, the store could run for example on an android device and install the applets directly from a phone — which makes sense as many applets are meant for use with smartphones.

- Scalability: the store main purpose now is to increase the number of smart card users to motivate smart card developers. In the distant future, many applets could be available and downloading the whole database might be inconvenient. But the store should always support at least partial offline operability.

- Updates: the store could define an API for applet updates: the applet and the client application would decide on shared secret based on JCAppStore identification (that the applet can trust to the client not to compromise its data).

Also, the store still contains discontinued Keybase support implementation. In case Keybase becomes more suitable to use as integrated software, the store could verify signatures more efficiently.

8 Conclusion

The JCAppStore is an operational card manager with good GUI design and intuitive use. The store offers selected applets from Enigma Bridge Curated list (see appendix B) along with descriptions, installation, use guide and links to more tutorials, documentation or source repository. While basic applet installation and distribution is simplified, the applet-specific parametrization (if used) still depends on a particular applet. For example, Ledger U2F applet installation and personalisation still require advanced smart card and cryptography knowledge (see appendix A), yet another applets installation is done using two clicks.

The card management is simple thanks to metadata management and routines automation. The errors caused by smart cards (unsupported SDK versions, invalid operational state, unknown Master key and much more) are handled either by advising on further steps that have to be taken in order to resolve the issue, or reporting a general error message in case the problem is difficult or impossible to resolve.

The software offered by JCAppStore is stored in a transparent GitHub repository. The software build is deterministic and thus the source code origin verifiable. The software integrity is verified using PGP signatures.

Bibliography

1. WANT, R. Near field communication. *IEEE Pervasive Computing* [online]. 2011, vol. 10, no. 3, pp. 4–7 [visited on 2020-03-27]. Available from: `https://ieeexplore.ieee.org/abstract/document/5958681`.

2. DOYLE, Ronald; HIND, John; PETERS, Marcia. *Smart card with integrated biometric sensor*. US. US Patent, 0095587.

6. *Card Specification V2.3.1* [online]. GlobalPlatform, Inc., 2018 [visited on 2019-01-29]. Available from: `https://globalplatform.org/specs-library/card-specification-v2-3-1/`.

7. *Secure Channel Protocol 03 Card Specification V2.2 – Amendment D* [online]. GlobalPlatform, Inc., 2009 [visited on 2020-03-25]. Available from: `https://globalplatform.org/wp-content/uploads/2019/03/GPC_2.2_D_SCP03_v1.0.pdf`.

8. DAVIS, A. Identifying and measuring quality in a software requirements specification. In: *Proceedings First International Software Metrics Symposium* [online]. Baltimore, MD, USA: IEEE, 1993, pp. 141–152 [visited on 2019-03-02]. Available from: `https://ieeexplore.ieee.org/abstract/document/263792`.

9. HORÁK, Jiří. *JCAppStoreContent* [`https://github.com/petrs/JCAppStoreContent`]. GitHub, 2019.

11. HORÁK, Jiří. *JCAppStoreParser* [`https://github.com/Aiosa/JCAppStoreParser`]. GitHub, 2020.

12. HORÁK, Jiří. *JCAppStore* [`https://github.com/crocs-muni/JCAppStore`]. GitHub, 2019.

13. ELLIS, B.; STYLOS, J.; MYERS, B. The Factory Pattern in API Design: A Usability Evaluation. In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 302–312. ISSN 1558-1225. Available from DOI: `10.1109/ICSE.2007.85`.

14. *Atomic Access* [online]. Oracle [visited on 2019-11-14]. Available from: `https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html`.

15. *Synchronized Methods* [online]. Oracle [visited on 2020-04-16]. Available from: `https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html`.

16. GRUSCHKA, N.; IACONO, L. L. Vulnerable Cloud: SOAP Message Security Validation Revisited. In: *2009 IEEE International Conference on Web Services*. 2009, pp. 625–631. ISSN null. Available from DOI: `10.1109/ICWS.2009.70`.

17. *Swing - JComponent* [online]. Oracle [visited on 2020-03-19]. Available from: `https://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html#paintComponent-java.awt.Graphics-`.

18. Jubula Automated functional testing [online] [visited on 2020-03-19]. Available from: `https://www.eclipse.org/jubula/`.

19. ORTIZ, C. Enrique. *An Introduction to Java Card Technology* [online]. 2003 [visited on 2020-04-16]. Available from: `https://www.oracle.com/technetwork/java/javacard/javacard1-139251.html`.

20. FOWLER, Amy. *Painting in AWT and Swing* [online]. Oracle [visited on 2020-03-19]. Available from: `https://www.oracle.com/technetwork/java/painting-140037.html`.

21. *Card Specification – ISO Framework Version 1.0* [online]. GlobalPlatform, Inc., 2014 [visited on 2019-03-23]. Available from: `https://globalplatform.org/wp-content/uploads/2014/03/GPC_ISO_Framework_v1.0.pdf`.

22. SVOBODA, Luděk. *Simulátor čipové karty v prostředí PC/SC* [online]. 2007 [visited on 2020-03-25]. Available from: `https://is.muni.cz/th/bjp9v/`.

23. PALJAK, Martin. *GlobalPlatformPro Wiki* [online] [visited on 2019-01-30]. Available from: `https://github.com/martinpaljak/GlobalPlatformPro/wiki`.

24. OPENSC. *OpenSC Wiki* [online] [visited on 2020-04-16]. Available from: `https://github.com/OpenSC/OpenSC/wiki`.

Resources

3. PHILLIPS, Kevin. *Credit card Signature Credit Free Photo* [*image*] [online] [visited on 2020-02-21]. Available from: `https://www.needpix.com/photo/1410285/creditcard-signature-credit-card-business-contract-financial-money-sign`.

4. OPENCLIPART. *SIM Card Vector Graphics* [*image*] [online] [visited on 2020-02-21]. Available from: `https://freesvg.org/sim-card-vector-graphics`.

5. WEBSTER, Tony. *Hardware Authentication Security Keys (Yubico Yubikey 4 and Feitian MultiPass FIDO)* [*image*] [online] [visited on 2020-02-21]. Available from: `https://commons.wikimedia.org/wiki/File:U2F_Hardware_Authentication_Security_Keys_(Yubico_Yubikey_4_and_Feitian_MultiPass_FIDO)_(42286852310).jpg`.

10. ICONS8. *Folder SVG image icon* [*image*] [online] [visited on 2018-11-28]. Available from: `https://icons8.com/icon/pack/free-icons/color`.

A An Example Of an Installation Process

This appendix shows an example of how the applet installation is performed using JCAppStore and GlobalPlatformPro. An extreme case was chosen to show what can or cannot be simplified.

A.1 Ledger U2F Applet Description

Ledger U2F is an applet that conforms to FIDO alliance standards[1], performing Universal Second Factor authentication. The open-source applet installation process includes parameters field described as follows[2]:

- 1-byte flag: provide 01 to pass the current Fido NFC interoperability tests, or 00,
- 2-byte length (big-endian encoded): length of the attestation certificate to load, supposed to be using a private key on the P-256 curve,
- 32 bytes: private key of the attestation certificate.

An example of a correct installation parameter value is

```
000140f3fccc0d00d8031954f90864d43c247f4bf5f
0665c6b50cc17749a27d1cf7664
```

Before using the applet, the attestation certificate shall be loaded using a proprietary APDU:

CLA	INS	P1	P2	LC	Data
F0	01	offset (high)	offset (low)	data length	Certificate data chunk

This command has to be repeated until the whole certificate is loaded onto the card. The currently loaded portion is specified in parameters P1 and P2. Example of a first APDU command:

[1] https://searchsecurity.techtarget.com/answer/How-can-U2F-authentication-end-phishing-attacks

[2] Source: https://github.com/LedgerHQ/ledger-u2f-javacard.

```
F0010000803082013c3081e4a003020102020a479012
80001155957352300a06082a8648ce3d040302301731
15301306035504031 30c476e75626279205069 6c6f74
```

A.2 Installation Process: GlobalPlatformPro

At first, the code has to be obtained from a repository and compiled. Let's assume we already have the .cap file. We need to install the applet using the following command:

```
$ gp --params 000140f3fccc0d00d8031954f90864
d43c247f4bf5f0665c6b50cc17749a27d1cf7664
--cap /path/to/ledger-u2f-applet.cap
```

Once installed, we need to find out the installed applet AID so we can send the personalisation commands. Following command

```
$ gp -l
```

returns the information about our card contents. SD or ISD is a security domain, APP is an applet and PKG a package. Since the card was empty before, we are looking for AID value in the form of `APP [AID]`:

```
Warning: no keys given, using default test key
404142434445464748494A4B4C4D4E4F
ISD: A000000003000000 (OP_READY)
     Privs:   SecurityDomain, CardLock,
     CardTerminate, CardReset, CVMManagement

APP: A000000617004F97A2E94901 (SELECTABLE)
     Privs:

PKG: A000000617004F97A2E95001 (LOADED)
     Version: 0.0
     Applet:  A000000617004F97A2E94901
```

Now we can send the certificates to the applet using `--apdu` command:

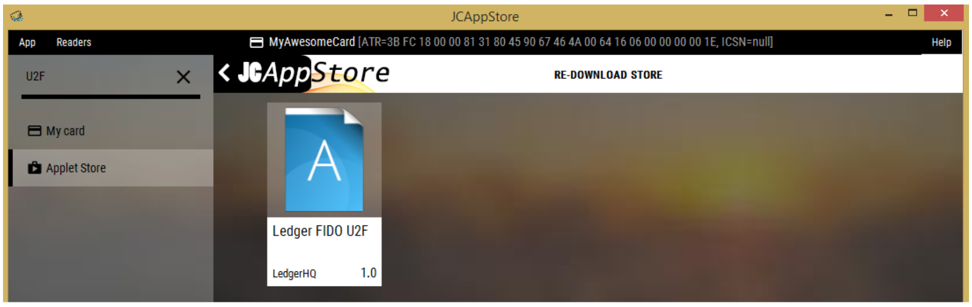
```
$ gp --applet A000000617004F97A2E94901
  --apdu F00100008030820l3c3081e4a0030...
```

A.2.1 Personalisation Data Acquirement

The user has to find out by himself how to generate the key and the certificate, using the applet description presented in previous section. Moreover, the personalisation commands expect raw binary data, whereas the key may be stored using DER format in ASN.1 notation, for example. See section A.3.1.

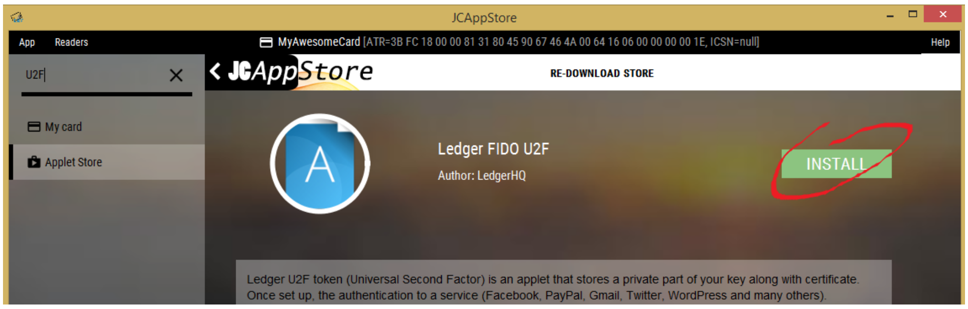
A.3 Installation Process: JCAppStore

First, the applet it searched for in the store. Once found, the applet is installed. In the end, the installed instance is selected and custom APDU sent.



MyAwesomeCard [ATR=3B FC 18 00 00 81 31 80 45 90 67 46 4A 00 64 16 06 00 00 00 00 1E, ICSN=null]

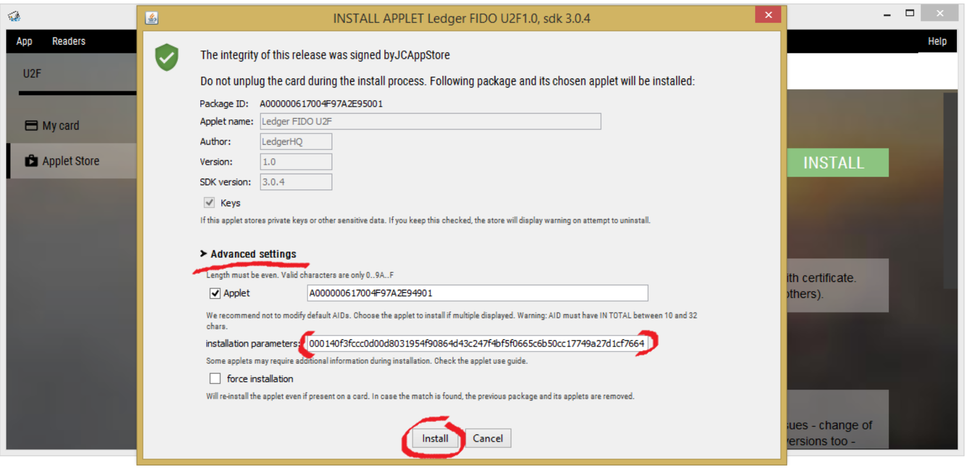
Figure A.1: Find the applet in the store.



MyAwesomeCard [ATR=3B FC 18 00 00 81 31 80 45 90 67 46 4A 00 64 16 06 00 00 00 00 1E, ICSN=null]

Ledger U2F token (Universal Second Factor) is an applet that stores a private part of your key along with certificate. Once set up, the authentication to a service (Facebook, PayPal, Gmail, Twitter, WordPress and many others).

Figure A.2: Open the applet details and select INSTALL.



INSTALL APPLET Ledger FIDO U2F1.0, sdk 3.0.4

The integrity of this release was signed byJCAppStore

Do not unplug the card during the install process. Following package and its chosen applet will be installed:

Package ID: A000000617004F97A2E95001
Applet name: Ledger FIDO U2F
Author: LedgerHQ
Version: 1.0
SDK version: 3.0.4

☑ Keys

If this applet stores private keys or other sensitive data. If you keep this checked, the store will display warning on attempt to uninstall.

Advanced settings

Length must be even. Valid characters are only 0..9A..F

☑ Applet A000000617004F97A2E94901

We recommend not to modify default AIDs. Choose the applet to install if multiple displayed. Warning: AID must have IN TOTAL between 10 and 32 chars.

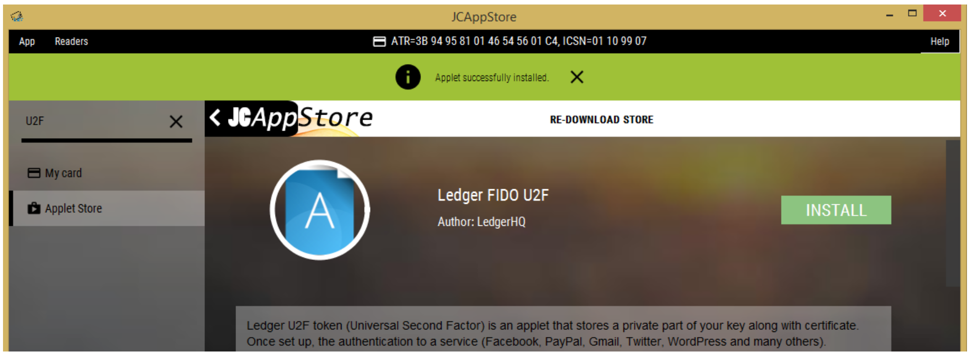
installation parameters: 000140f3fccc0d00d8031954f90864d43c247f4bf5f0665c6b50cc17749a27d1cf7664

Some applets may require additional information during installation. Check the applet use guide.

□ force installation

Will re-install the applet even if present on a card. In case the match is found, the previous package and its applets are removed.

Figure A.3: Open advanced settings and provide installation parameters. Click install and wait.



JCAppStore

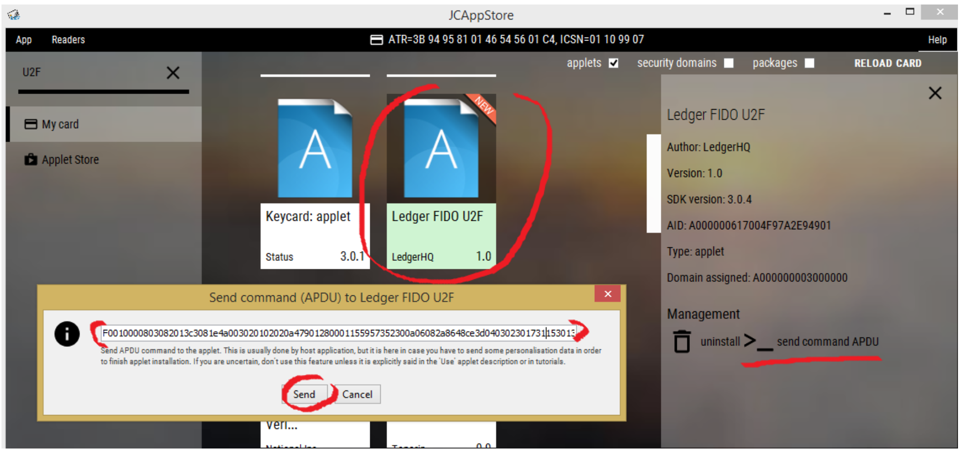
ATR=3B 94 95 81 01 46 54 56 01 C4, ICSN=01 10 99 07

Applet successfully installed.

Ledger FIDO U2F
Author: LedgerHQ

Ledger U2F token (Universal Second Factor) is an applet that stores a private part of your key along with certificate. Once set up, the authentication to a service (Facebook, PayPal, Gmail, Twitter, WordPress and many others).

Figure A.4: The installation was successful.



JCAppStore

ATR=3B 94 95 81 01 46 54 56 01 C4, ICSN=01 10 99 07

Ledger FIDO U2F
Author: LedgerHQ
Version: 1.0
SDK version: 3.0.4
AID: A000000617004F97A2E94901
Type: applet
Domain assigned: A000000003000000

Send command (APDU) to Ledger FIDO U2F

F0010000803082013c3081e4a003020102020a47901280001155957352300a06082a8648ce3d040302301731153013

Send APDU command to the applet. This is usually done by host application, but it is here in case you have to send some personalisation data in order to finish applet installation. If you are uncertain, don't use this feature unless it is explicitly said in the 'Use' applet description or in tutorials.

Figure A.5: Find the installed instance on your card and send the certificate.

A.3.1 Personalisation Data Acquirement

The "Use" applet description would contain following information on how to acquire the personalisation data using OpenSLL.
At first, the key must be generated (p256 elliptic curve).

```
openssl ecparam -name prime256v1 -genkey -out
   key.pem
```

However, the `key.pem` file contains more data — not only the private key, and the format is not a raw binary. Parse the key into hexadecimal readable form by using

```
openssl ec -text -in key.pem -out hexkey.txt
```

The `hexkey.txt` file will contain similar output to:

```
Private-Key: (256 bit)
priv:
  19:c6:b3:af:81:ab:55:d2:9b:e7:4b:0c:fe:
  ba:49:3b:93:64:39:6d:60:d4:f5:32:37:7d:
  54:3b:97:e3:cf:82
pub:
  04:14:0a:91:b8:dc:90:10:06:14:57:b1:a8:
...
```

We need the data in the "priv" section. Next, create the certificate, providing the number of days it should be valid for:

```
openssl req -new -sha256 -key key.pem \
  -out csr.csr
openssl req -x509 -sha256 -days [your value] \
  -key key.pem -in csr.csr \
  -out certificate.der -outform DER
```

The certificate has to be used in a hexadecimal string representation, and sent to the card. The description would also provide further example of uploading this data onto the installed applet and basic example of how to get started with the applet usage.

B Smart Card Applets Review

The Curated List of JavaCard Applications[1] contains a list of applets and other software tools related to smart cards. From this list, applets were selected that:

- are likely to be useful (no educational, proof of concept implementation),
- are kept up to date (not discontinued projects),
- have a client application that can handle the communication.

The applets had to be investigated in the following manner. The description is in `[stage].Decline:[reason]([example])` form.

1. Identify the applet potential use. Decline: significal problems (missing repository).
2. Compile the applets, identify software requirements. Decline: compilation problems (use of commercial libraries).
3. Asses the applets. Decline: implementation or use problems (obsolete SIM-toolkit technology).
4. Identify host applications, compile if source code provided only. Decline: missing or problematic (proprietary API).
5. Install the applet. Decline: complex procedure or errors (`0x6881` logical channel not supported when personalising).
6. Use the applet. Decline: errors in usage or missing tutorials (NDEF smartphone applications are unable to work with an NFC tag applet).

Following tables summarize the results. Compilation issues section contains information on how the source code had to be modified in order to compile (and thus building from the source will not result in the same `.cap` file).

[1] Available on `https://github.com/EnigmaBridge/javacard-curated-list`.

B.1 Accepted Applets

Applet	Description	Repository
JMRTD Machine readable travel documents	Implementation of MRTD standards (ePassport).	LINK
JMRTD without EAC	Modification of the above without EAC (Extended Access Control). For cards that do not support EAC.	LINK
Yubikey Oath	Applet that generates one time passwords for 2FA authentication.	LINK
IsoApplet	Storing private keys and performing asymmetric cryptography operations using PKCS#15 standard	LINK
HOTP via NDEF	One time password generator delivering the data into NDEF tag (e.g. an URL address).	LINK
GIDS applet	Generic Identity Device Specification, supported natively in Windows (certutil - keys and certificates management).	LINK
KeepasNFC	An applet that offers to store a password of Keepas database on a smart card.	LINK
Yubico OpenPGP	A PGP-compatible smart card. Offers key creation and from-card use.	LINK
SmartPGP	A PGP-compatible smart card. Offers also a private key import unlike OpenPGP.	LINK
FluffyPGP	Another PGP-compatible smart card.	LINK
JCOpenPGP	And yet another PGP-compatible smart card.	LINK
StatusKeyCard	A bitcoin hardware wallet implementation, usable with WallEth (android) app or Gnosis Safe (both android and iOS).	LINK
SatoChip Applet	A (beta) bitcoin hardware wallet implementation (BIP32 support).	LINK
JCAlgTest	Applet for smart card capabilities and functionality support testing.	LINK
JCMemory	Applet for measuring smart card free persistent memory, a subset of JCAlgTest.	-

"SatoChip" applet had to be modified due to a missing `short` cast.

B.2 Applets for Further Investigation

We were unable to get these applets working. However, the applets have potential. Some require a host application to be used with, other slight modifications. Some applets are listed here because there are other variants with better features available (and to resolve the issue was not a top priority).

Applet	Issue
YkOtp Applet	host application works with YubiKeys only (seems the reader vendor is verified)
OpenFIPS 201 PIV	Personal Identity Verification, will add once a suitable host found and tested
PivApplet	Personal Identity Verification, will add once a suitable host found and tested
Ledger U2F	unable to initialize the applet: 0x68 81 error
U2FToken	based on Ledger U2F, we had the same 0x68 81 error
SIM password wallet	we could not install the host application due to missing OpenMobileApi library, the host application can be updated to remove this library dependency and use android API instead (included API v28), unfortunately we do not have a device with this or newer android version and were unable to test it
JC Password Manager (StegoPass applet)	as a project developed during a university course, the assessment is delayed — these projects tend to be discontinued after the course ends
OpenNDEF full applet	though this applet was first included, some API changed and smartphone host applications are no longer able to use this applet, awaits a fix
Secure bitcoin wallet	the applet use has to be tested and a quick tutorial created

B.3 Excluded Applets

Applets that are experimental, outdated, missing host applications or not fit for real personal usage (for example a problem domain complete solution consisting of many components, usually including even server implementation).

Applet	Reason
Fake Estonian EID	the applet purpose is educational
Electronic driver licence	not usable applet, not supported by authorities
Security In Computing (SIC) Applet	more educational implementation, alternative to EstEID card, a server must be run
FedICT Quick-Key Toolset Project	unable to issue `SELECT` command
IdentityCard applet	repository was removed
Belgian EID	educational rather than practical use, a server must be run
SShSupport applet	missing information on AID value, a part of uClinux (outdated)
MuscleApplet	outdated project
CoolKey Applet	MUSCLE applet, outdated
ORWL KeyFob-applet	unlocks an ORWL computer, useful for an ORWL computer owner, provided with the device
PKCS#15 applet	repository was removed
PKI applet	repository was removed
Cryptonit applet	experimental, contains a subset of PIV specification
CCU2F Fido	based on Ledger U2F, this applet offers the same functionality with minimum support of SDK 3.0.5 — we were unable to test it
Yubisec	unable to use with Yubico host applications, refusing other hardware than YubiKeys
Biometric Authentication	experimental project without host application
OneCard	project not finished, stagnating since 2015
OTP: One Time Password	proof of concept implementation based on obsolete SIM-toolkit technology

TIM: Trusted Identity Module	not fit for personal use
OpenEMV	educational implementation, EMV payment cards cannot be issued by users themselves
Simple Wallet	smart card applet demonstration with three basic commands (withdraw/store/get balance)
EPurse	without a host app, suitable for issuing customer membership cards for example
AppSecure	applet useful to developers, if supported an application requires the card to be inserted in order to start (software piracy control)
PBOC3 Applet	similar to OpenEMV, PBOC3 is national Chinese bank smart card standard compatible with EMV
PayPass	using simply tap (discontinued platform), Mastercard-compatible implementation: nothing a user would issue himself
SimplyTapp	discontinued platform, set of card examples and compatible drivers to use with
JavaCard Wallet	only basic functionality, similar to Simple Wallet (withdraw/store/get balance/pin)
EMV-CAP Applet	EMV-CAP emulation, proof of concept for an article
Mobile banking stk	based on obsolete SIM-toolkit technology
Loyalty Applet	hardcoded data/paths in host applet — educational implementation
CryptSetup javacard	proof of concept implementation
SmartCardNFC applet	just a copy of original repository (KeepasNFC)
SimPasswordManager	"Requires associated Java Card applet to be loaded in the SE (SIM card or embedded SE device) in advance." (`https://github.com/nelenkov/sim-password-manager`)
Sim Password Store	based on obsolete SIM-toolkit technology
TrueCrypt	Truecrypt support for javacard platform, discontinued (reason: http://truecrypt.sourceforge.net/)
PassMsg	not a serious project (vulgar error messages)

Myst	not fit for personal use, a project that demonstrates a javacard technology usage for "distributed key generation, signature, and decryption with private key distributed among multiple separate entities" (`https://github.com/OpenCryptoProject/Myst`)
SigAnima	the applet is not supported by any host app (proprietary API)
Virtual Keycard applet	this project actually does not contain any applet (just a copy from another repository), the aim is to run the applet virtually inside a smartphone
Electronic Health card	unable to find a host application
HealthCard	described as a prototype by authors
OpenNDEF stub applet	a NDEF applet version that is not yet docummented, requires a backend service implemented by another applet
OpenNDEF tiny applet	a subset of OpenNDEF full applet that suppor only NDEF tag reading, the data is loaded on the applet when installing (hexadecimal form)
NDEF javacard applet	cannot be deployed from the store, provides compile-time only data writeability
PICO NDEF applet	repository was removed
Ledger Hardware Wallet	the project was discontinued
BitcoinWallet	the project is long inactive - no comments, no commits, issue tickets not answered
LamassuCard	the project seems to be educational – the applet only supports ECDSA signature and public key export commands, the applet would be accepted if the interface provided more functionality or the host application would be more user-friendly

C Source Code

The source code is provided with the thesis. The store and the store content are both available on GitHub. The store is licensed under MIT.

C.1 JCAppStore

The repository[12] contains following folders, mostly documented using `README` files:

1. `deterministic-build` - information on how to generate the same binaries
2. `graddle/wapper` - gradle automatic build tool files
3. `installer-unix` - scripts for tarball generating and launcher script
4. `installer-win` - files to create an IzPack windows installation wizard
5. `readme-res` - resources for top-level `README.md`, also contains the thesis
6. `src` - source code
7. `test` - jubula test suite

C.2 JCAppStoreContent

The repository is the source content downloaded by JCAppStore[9]. This repository contains folders:

1. `JCAppplets` - `.cap` files and signatures,
2. `Resources` - images,

and files:

1. `info_[lang].json` - main store file in `lang` language,
2. `store.asc` - public key for the signatures verification.