Hardware Security JavaCard Design Document
Group 3

Matti Eisenlohr
S1000794 (RU)

Egidius Mysliwietz
S1000796 (RU)

Alessandra van Veen
S4683382 (RU)

Laura Philipse
S4528751 (RU)

June 4, 2021

Contents

1 Introduction

We designed a smartcard application to be used for car rental services. This application allows a company to give out smartcards to their customers, who can then use those cards to rent a vehicle at a terminal. The smartcard communicates with the car to make sure the customer indeed has access to this car, and to keep track of the kilometerage. When the customer wants to return the car, they can return the car and pay at a terminal.
In this document, we first investigate the use cases, threats and security requirements needed for such an application. We then state our major design decisions and explain the protocols used.

2 Use cases

Use Case 1: User rents a car
Actor: User
Description: This scenario describes the situation where a user rents a car and gets access to use the car with their card.
Basic Flow:

1. The user requests to rent a car at the company
2. The user inserts the card into a company terminal.
3. The company terminal determines whether the card is valid.
4. The company terminal updates the card with the ignition key and the kilometerage.
5. The company terminal gives a confirmation message.
6. The user takes the card out of the company terminal.

Alternative Flow 1: User does not yet own a card

1A1 The user requests to rent a car at the company.

1A2 The company issues a card to the user.

Alternative Flow 2: Invalid card

3A1 The company terminal determines whether the card is valid.

3A2 The company terminal deems the card as invalid.

3A3 The company terminal gives an error message.

Use Case 2: User uses a car
Actor: User
Description: This scenario describes the situation where a user enters the car and uses it for a period of time.
Basic Flow:

1. The user gets in the rented car.
2. The user inserts their card into the card terminal.
3. The car terminal verifies the card and starts the engine.

4. The user drives the car.
5. The car continuously updates the card with the kilometerage.
6. The user stops the car.
7. The user takes the card from the terminal.

Alternative Flow 1: The card is invalid.

3A1 The car terminal verifies the card.

3A2 The car terminal deems the card invalid.

3A3 The car terminal displays an error message.

Alternative Flow 2: Card is taken from terminal while driving

4A1 The user drives the car.

4A2 The user takes the card from the car terminal.

4A3 The car terminal displays an error message.

4A4 The car slows down until it stops.

Use Case 3: User returns a car
Actor: User
Description: This scenario describes the situation where a user returned the car to the company. The company reads out the card for its kilometerage and removes the ignition key.
Pre-condition: The user has rented a car from the company.
Basic Flow:

1. The user inserts their card into the company terminal.
2. The company terminal verifies the card.
3. The company terminal reads out the kilometerage on the card.
4. The company terminal removes the ignition key and kilometerage from the card.
5. The company terminal gives a confirmation message.
6. The user takes their card from the company terminal.

Alternative Flow 1: Invalid card

2A1 The company terminal determines whether the card is valid.

2A2 The company terminal deems the card as invalid.

2A3 The company terminal gives an error message.

3 Threat modelling

We consider the following threats:
An attacker can commit a man-in-the-middle attack on the traffic between the smartcard and the terminal. They could want to change the kilometerage, either to a lower value so they have to pay less, or to a higher value if the attacker is part of the company so they would get more money. They might also want to duplicate a card to use that card to rent a car without paying for it.

A customer could also remove the card from the terminal or the car while the protocol is still in process, causing possibly unexpected behaviour. They could try to manipulate the kilometerage or try to maintain data that should be deleted from the card.

An attacker could set up a fake endpoint, controlling the inputs to the car or card. They could use this to change the kilometerage or to possibly hijack the car.

We consider malicious cars to be outside our scope, as are customers claiming that their card got stolen when it did not, to avoid having to pay for the kilometers driven. We also assume that the back-end database cannot be attacked in any way.

4 Security requirements

SR1 **Non-repudiation of the rentals of a customer:** Any customer should not be able to deny the fact that they signed the rental contract and used a specific smart card with a specific car for a specific period of time.

SR2 **Correctness/Integrity of kilometerage:** When renting a car, the number of kilometers driven during the current rental lease should be accurately reflected on the smart card, within some small margin of error, as updates to the card may not be continuous. Take note that this property only applies to the card, not the car. Hence, the card needs to reflect the accurate number of kilometers driven, not the number reflected by the car, as it may have been tampered with.

SR3 **Mutual authentication between card and terminal:**

- SR3.1 Before receiving a car and when returning that car, a customer will insert their card into a terminal. Before the card's data is processed, the terminal needs to know that the card is who it claims it is.
- SR3.2 At the same time, a card should block its service if it detects the terminal to be illegitimate, meaning it is not an actual terminal by the rental company.

SR4 **Mutual authentication between card and car:** During each rental period, a car can only be unlocked by a single card. Before starting, the car needs to check if the card used is part of the respective rental contract.

5 Major design decisions

DD1 **Online status of car:** The car does not need to be online at any point.

DD2 **Online status of terminals:** The rental terminals at the reception desk as well as those at airports and other locations are assumed to be always online.

DD3 **Keys between card and reception desk terminal:** As there are a lot of cards (and potentially many reception desks in sister locations), we decided to use asymmetric cryptography.

DD4 **Rental contracts:** A central database exists listing all card to car assignments. From this database, the car receives the relevant information about the card, such as the unique identity number and the public keys of the card.

DD5 **Card material:** To prevent cards from breaking (and subsequent problems due to stranded customers), the cards are made from a sturdy, robust material.

DD6 **Unlocking the car:** The car will be unlocked by a wireless car opener (a physical device) when parked and by a button on the inside of the car.

DD7 **Central Database:** A single database storing all rental information about all customers as well as the public keys of all cars and cards exists and is assumed to always be up to date. This database is query-able and gives out information in the form of trusted certificates about IDs and private keys.

DD8 **Certificate verification:** Every car and card store the public signing key of this database on them.

DD9 **Inspection of logs:** When a car is returned, an employee at the return location manually checks the logs stored on the car.

DD10 **Existing keys and IDs:** Terminals and cards can already have a certificate and an ID assigned to them.

DD11 **Blocking cards:** In practice, when you want to block a card, an employee is called who verifies your identity and decides if you're trustworthy enough to have your card blocked.

6 Overview of credentials

In our protocols we use the following credentials and notations:

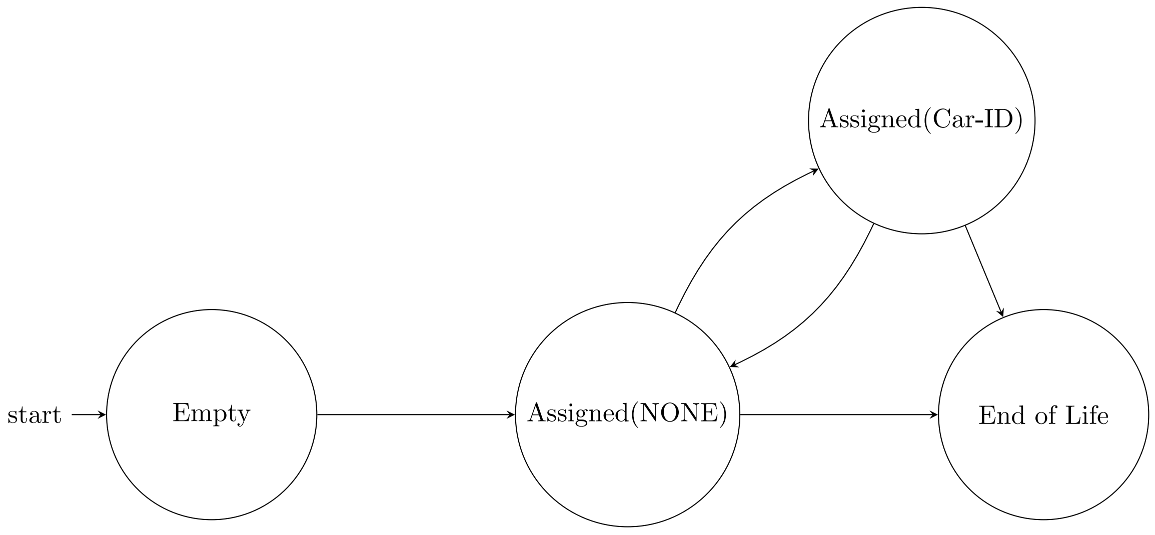
- $pubsk_{card}$ = public signing key of the smartcard
- $privsk_{card}$ = private signing key of the smartcard
- $pubsk_t$ = public signing key of the reception terminal
- $privsk_t$ = private signing key of the reception terminal
- $pubsk_{car}$ = public signing key of the car
- $privsk_{car}$ = private signing key of the car
- $pubsk_{db}$ = public master signing key of the company's central database to verify certificates issued by the database
- $privsk_{db}$ = private master signing key of the central database for issuing certificates

Notation

- Card-ID = ID-number of the smartcard (unique for each smartcard)
- Car-ID = ID-number of the car (unique for each car)
- Term-ID = Id of a specific reception terminal (unique for each terminal)
- h(m) = hash of message m
- $\{m\}_k$ = message m encrypted with key k
- m = message m sent over communication channel
- {m, n} = messages m and n sent over communication channel
- manipulation-bool = boolean value on the card to indicate if there was an attempt to lower the milage
- $signed_K(m)$ = m ++ signature(h(m),K): message m signed with key K in the format shown here
- $CERT_x = signed_m sk$(m): Certificate x of message m signed with master key msk
- $CERT_{card} = signed_{privsk_{db}}(pubsk_{card}, \textit{Card-ID})$
- $CERT_{term} = signed_{privsk_{db}}(pubsk_t, \textit{Term-ID})$
- $CERT_{car} = signed_{privsk_{db}}(pubsk_{car}, \textit{Car-ID})$
- successBYTE = 0xFF: Confirmation that last regular protocol step has succeeded

7 Detailed protocols

The smartcard has the following lifecycle with these states and transitions:



"Empty" describes the state, in which the smartcard does not have a user assigned to it.
"Assigned(NONE)" describes the state in which the user information is stored on the card, but it is not assigned a car.
"Assigned(Car-ID)" describes the state in which the card is assigned to a car.
"End of Life" describes the state in which the card no longer functions, either because it has been blocked or because it reached its expiry date.

P1 **Mutual authentication between smartcard and car for ignition:** The smartcard and car authenticate themselves to the other party via a challenge-response protocol. The smartcard already has the certificate of the car, it got it from the terminal.

1. Card gets inserted into the car terminal
2. Card $\rightarrow$ Car: $CERT_{card}, Nonce_{card}$
3. Car verifies the certificate
4. Car $\rightarrow$ Card: $signed_{privsk_{car}}(Nonce_{card}), Nonce_{car}$
5. Card verifies signature with $pubsk_{car}$ and only proceeds if signature checks out
6. Card $\rightarrow$ Car: $signed_{privsk_{card}}(Nonce_{car})$
7. Car verifies signature with $pubsk_{card}$ and logs successful or unsuccessful authentication attempt
8. Car $\rightarrow$ Card: $signed_{privsk_{car}}(successBYTE, Nonce_{card} + 1)$
9. Car starts if authentication was successful

P2 **Mutual authentication between smartcard and reception terminal:** The smartcard and reception terminal authenticate themselves to the other party by verifying each other's certificates over a challenge-response protocol. During this protocol the smartcard temporarily stores the public signature key of the terminal (the terminal received it from the central database beforehand), which gets removed from the card once it is no longer inserted into the terminal.

1. Card gets inserted into terminal
2. Card $\rightarrow$ T: $CERT_{card}, nonce_{card}$
3. Terminal checks whether the card is blocked.
   (a) If blocked: T $\rightarrow$Card: End of Life.
   Card transitions to the state "End of Life". The protocol ends.
   (b) If not blocked: proceed as usual.
4. Terminal checks validity of certificate
5. T $\rightarrow$ Card: $CERT_{term}, nonce_{term}$
6. Card verifies validity of certificate
7. Card $\rightarrow$ T: $signed_{privsk_{card}}(nonce_{term})$
8. Terminal verifies signature and nonce
9. T $\rightarrow$ Card: $signed_{privsk_t}(successBYTE, nonce_{card})$
10. Card verifies signature and nonce

P3 **Assignment of car to smartcard:** The smartcard requests a car at the terminal and sends along a sequence number, signed with the card's private signing key. If the sequence number is correct, the terminal creates a database entry and sends the certificate of the car along with a sequence number, signed with its private signature key.

1. Authentication protocol (P2)
2. Card $\rightarrow$ T: $signed_{privsk_{card}}$(“Assign me a car”, $nonce_{term} + 1$)
3. Terminal checks if $nonce_{term} + 1$ is correct
4. Terminal adds database entry with Car-ID and Card-ID
5. Terminal displays the Car-ID
6. T $\rightarrow$Card: $signed_{privsk_t}(CERT_{car}, nonce_{card} + 1)$
7. Card verifies the certificate and the sequence number
8. Card transitions to state “Assigned(Car-ID)” and stores the car data from the certificate
9. Card $\rightarrow$T: $signed_{privsk_{card}}(successBYTE, nonce_{term} + 2)$
10. The terminal verifies the signature and displays a message to let the user know they can remove the card now
11. Card gets removed from terminal

P4 **Car return and kilometerage check:** This protocol incorporates a challenge-response and a sequence number to prevent replay attacks. All messages are signed by their author/sender.

1. Authentication protocol (P2)
2. Card $\rightarrow$ T: $signed_{privsk_{card}}$(“Car return”, $nonce_{term} + 1$, *manipulation-bool*)
3. Terminal verifies the signature and sequence number and checks if the manipulation-bool is false, if not, a human verifier is needed
4. T $\rightarrow$ Card: $signed_{privsk_t}(nonce_{kilometerage}, nonce_{card} + 1)$
5. Card verifies signature and sequence number
6. Card $\rightarrow$ T: $signed_{privsk_{card}}(kilometerage, nonce_{kilometerage}, nonce_{term} + 2)$
7. kilometerage is reset to 0 on the smartcard and $CERT_{car}$ gets removed from the smartcard
8. Terminal verifies signature, nonce and sequence number
9. Card transitions to state “Assigned(NONE)”
10. T $\rightarrow$ Card: $signed_{privsk_t}(successBYTE, Nonce_{card} + 2)$
11. Card gets removed from terminal

P5 **Adding kilometerage to smartcard:** The kilometerage is continuously updated during the drive every time the kilometerage changes on the car display. The car sends the current kilometerage to the card, which saves it and sends it back, signed with its signing key. The car verifies the signature and logs the event, so that the kilometerage can later be verified.

1. Authentication protocol (P1) (only at first iteration)
2. Car $\rightarrow$ Card: $signed_{privsk_{car}}(current\_kilometerage)$
3. Card verifies the signature and checks if this new value for kilometerage is actually bigger than the value currently stored on the card
   (a) If not: kilometerage will not be updated and card sets a boolean flag manipulation-bool to true to show there might have been manipulation.

(b) If yes:

i. Card saves kilometerage.

ii. Card $\rightarrow$ Car: $signed_{prisk_{card}}(CONFIRMATION_{BYTE}, current\_kilometerage)$

iii. Car checks signature and logs the event (timestamp and kilometerage).

4. If the message 2 gets interrupted and the car does not receive message ii from the card within a set amount of time, the car logs that the card did not receive the update

P6 **Card blocking:**

1. Customer indicates they want to block their card.
2. Customer gives Card-ID to employee.
3. Employee selects card to be blocked.
4. Database deletes Card-ID from table of active cards.
5. Employee confirms to customer card has been blocked.

8 Implementation guide

The implementation can be found at https://github.com/emysliwietz/hardware_security/

8.1 Dependencies

Smartcard

- As our keys and certificates are rather long, we opted to use the ExtendedAPDU interface. It wouldn't be impossible to change this, but it would require some work and, more importantly, would cause a lot of method overhead that don't make the change an improvement.
- The smartcard used to use the java native interface ByteBuffer class to write some data to the APDU (but not for reading). In essence, this class does the same as Util.putShort, but is a little easier to work with as it manages offsets itself. As this class is implemented via the native i/o library, it should not create any overhead for the smartcard. Using these ByteBuffers necessitates a fairly recent version of Java. Any version above 10 should work. We decided to not have any not strictly necessary libraries that don't help getting closer to our protocol descriptions above. Therefore, the smartcard and the Communicator interface do not use ByteBuffers, but Terminals do for ease of use.

Terminals For communication between the terminal and the database, we decided to use a Stack to simulate the messaging channel (as implemented in `Receivable`). As a terminal and a database communicate together at the same time, our implementation uses multiple threads to simulate these two instances.

GUI The GUI is written in JavaFX, therefore requiring the JavaFX library. There isn't any inherent reason not to use Swing, we simply opted for JavaFX as it is a newer library and because JavaFX looks nicer. It might be necessary to include some of the JavaFX modules in the VM options to get the GUI to run properly (`--module-path <path-to-JavaFX>/javafx-sdk-11.0.2/lib --add-modules=javafx.controls,javafx.graphics,javafx.web,javafx.base,javafx.fxml, javafx.media,javafx.swing`) (Out of those modules, probably only controls, graphics, base and fxml are needed).

Database The database stores data in hexadecimal string representation in an SQLite database file. To enable this interaction, we use the JDBC.jar module version 3.34.0. In a real use case, using a single SQLite database file would not scale well. There aren't any SQLite-specific features we're using, so porting the application to a more scalable database like MariaDB would be fairly simple. We opted for SQLite as it allows us to have a contained implementation without the dependency of an additional database server.

8.2 Implementation details

General

- Certificates or any other forms of signed hashes are send by first prepending their length. In practice, this length is always 64 due to the `sign` algorithm utilized. We could change our protocol to use a global constant instead of sending these lengths every time we send a signed hash. However, some public cryptography implementations could have variable length signatures, so we decided that the additional overhead of a few bytes extra per hash is worth the added flexibility.
- For converting our keys to and from bytes, we opted for a consistent keylength instead, as defined in `ProtocolComponentLengths`. Therefore, the binary representation of our keys starts with the length of the exponent in bytes, then contains the exponent zero-padded to 64 bytes, followed by the length of the modulus and the modulus zero-padded to 64 bytes. Public keys usually have significantly smaller exponents compared to private keys, but it is a nicer implementation in our protocols to pad them to the same size.

Smartcard

- Smartcard makes use of an interface called `Communicator` to handle some utility functionality and define APDU codes (CLA, INS, SW) as well as any kind of global methods both smartcard and other terminals use. We're using the interface `default` keyword to circumvent Java's multiple inheritance non-feature. This is just to prevent duplicated code, no difference would arise if the code would be copied into the `Smartcard` class.
- A Smartcard is passive in the sense that it cannot initiate any protocols. For cryptographic reasons, in our protocols above, we decided to make the smartcard send a message first. We implemented these protocols by first sending an APDU with empty data from the terminal to the card to select the right protocol. The response by the smartcard is then the "real" first message as specified in our protocols above.
- The smartcard uses `javacard.security.RandomData` to generate random data, for use in nonces and IDs. In the simulator, this method always produces the same random data, but as mentioned on brightspace, this shouldn't be an issue in practice.
- In `jcardsim 3.0.4-snapshot`, the version we're using, RandomData seems to be outdated compared to the documentation. Changes that would need to be made to port to a newer version are described in `CryptoImplementation`. To state them here, `RandomData.ALG_SECURE_RANDOM` should be replaced in favor of `RandomData.ALG_SECURE_RANDOM`, at the very least for nonce creation. There is some argument to be made to change it to `ALG_FAST` for ID creation, but the performance improvement isn't worth having another RandomData instance. Further, `generateData` would need to be replaced with `nextBytes`, similar to `SecureRandom`.

- Most data in the smartcard is transient, meaning it clears itself when the card is powered of (disconnected from a terminal). For values that are not `byte[]`, these values are saved. Our card uses a `SmartcardCrypto` class to handle cryptography, meant to simulate a trusted secure enclave. On each reset, this class is cleared from memory. Therefore, we need to store the variables `cardID, cardCertificate, privateKey` in a non-transient-fashion. Doing this via `JCSystem` did not work, so instead, we call `new byte[]`. Should a better implementation exist, this can easily be changed by re-implementing `newStaticB` in `Communicator`.

- Most of our protocols consist of multiple exchanges between smartcard and terminal. We decided to add a `CLA`-byte for continuation messages in our protocols as well as respective `INS`-bytes for every message. This is a somewhat clunky implementation, but it enables us to have some more control over protocol flows, in case a protocol is interrupted.

- Since we try to initialize the smartcard with more than 255 bytes of data, we could not use the regular install method for initializing the card. To fix this issue, we created another protocol, where the database sends the data to the reception terminal which forwards the data to the next uninitialized smartcard that is inserted into it. In a real life scenario, the initialization process would be performed by an employee, before such a card is given to a customer.

- Regarding RAM usage, the smartcard should not use more than 2 KB at any point (assuming that local variables in methods get removed by the Java Garbage Collection). The by far most RAM intensive method is init(), which receives roughly 470 bytes of data. It might be that all these values might be saved twice since we use our memCpy method to copy them to local variables. Adding to that the global variables of Smartcard, which should be less than 1 KB, the smartcard should, even at its worst point, still use no more than 2 KB.

Terminals

- All terminals (receptionTerminal, auto and database) inherit from an abstract class `CommunicatorExtended`, which serves much the same function as `Communicator`, which it "implements" (quotes because actual implementations are in the interface using defaults). Notable differences are additional methods not needed by the Smartcard, the usage of the `Receivable` interface described in Section 8.1 and the replacement of `JCSystem` methods with non-smartcard features (such as `new byte[]` in favor of `JCSystem.createTransientByteArray()`, for example).

- Analogous to the above difference, the smartcard uses `CryptoImplementation` for its cryptographic functions, whereas the terminals use `CryptoImplementationExtended`. These contrast only in the usage of `SecureRandom` from `java.security` instead of `RandomData` from `javacard.security`, which has the added benefit of actually producing non-repeating random data.

8.3 Discussions, shortcomings and potential improvements

- We use `RSA-512` keys in our protocols. Switching to `ECC`-keys would make our key sizes significantly smaller and would be advised if used in practice. We decided not to do this for now as it is not as well supported in our version of `jcardsim`. In a similar manner, our signatures could be changed from `Signature.ALG_RSA_SHA_PKCS1` to make them much smaller.

- As long as signatures and hashes have a fixed size, which they most likely do, there is no need to send their length in our protocols. This reduces flexibility somewhat, but depending on the use case, it might be worth it. Changing this means going over every protocol, but it shouldn't be too difficult as we already use a short to manage offsets.

- `ExtendedAPDU` isn't supported by some physical smartcards, so it might be advisable to extend protocols to send data sequentially instead of at once. This would need significant changes to the protocols as well as a lot more APDU bytes, but is definitely doable, should the added message sending overhead be worth the extensive terminal support.

- We very rarely if at all use `javacard Util` methods, instead implementing them ourselves. This allows us to create methods with a similar pattern that the `Util` class does not support and makes our implementation easier. Should `Util` methods somehow turn out to be much faster, changing our implementations to use `Util` instead would be fairly easy.

- There are some places where we use `int`s in the smartcard. Assuming that some smartcards don't support this data type, this is problematic. We originally used them to experiment with lengths before we decided on the cryptographic algorithms to use. Since they're only used sporadically, we deemed it not worth the effort of changing and potentially debugging all protocols just to gain a few more bytes of memory savings, but in theory, using `short`s or in some cases even `byte`s instead would be much preferable. Since all lengths and offsets are defined in `ProtocolComponentLengths`, all that would need to be done is change `INT_LEN` to `SHORT_LEN` and `getInt` and `putInt` to `getShort` and `putShort`.

- We opted for an ID length of 5 bytes. In practice, that's much larger than necessary, but we picked the value to reduce collisions when generating ID cards, as such a collision could lead to problems (namely, the system treats those cards as identical or might even crash in the worst case, although that's unlikely). Changing the length of an ID is as easy as changing the value in `ProtocolComponentLengths`. No other changes should be necessary.

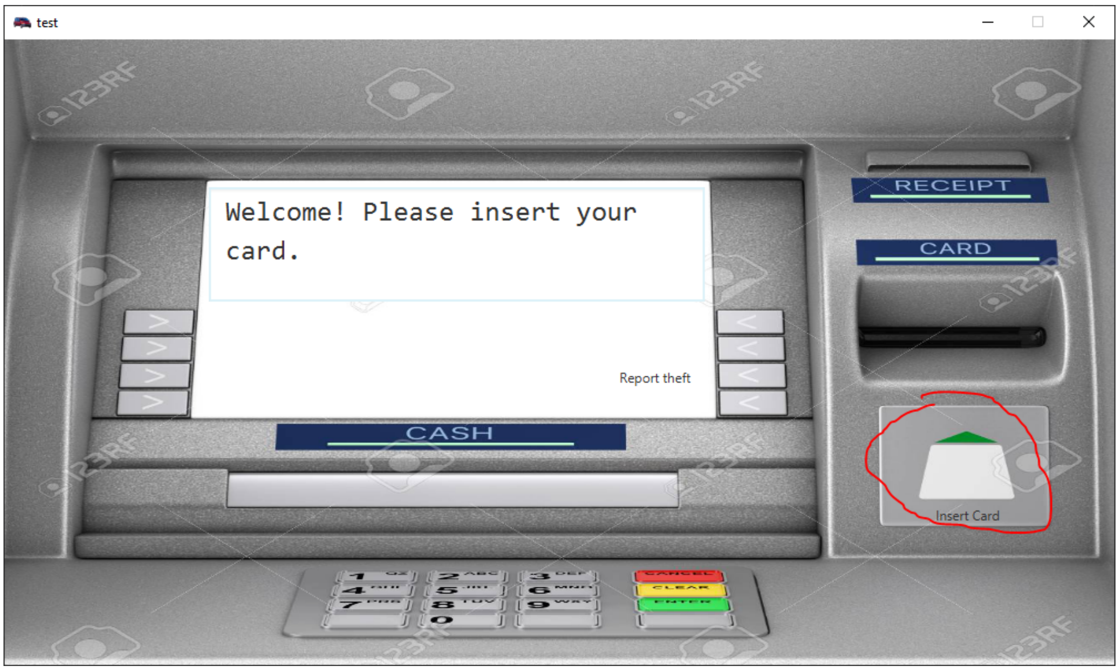
- In some method signatures, such as `memCpy`, the original method uses `shorts` to specify offsets, but we also declared messages using `ints` as parameters and calls the original method, which saves us the effort of casting them manually every time. This shouldn't make any difference in practice, as the value might be computed as an int but subsequently converted to a short either before or after method invocation. However, depending on how a smartcard CPU works, this might present an issue. Using AMD 64-bit calling convention, each argument would be passed using a register, which are 64bits in length, so it makes little difference if we pass an integer or a short. Should the JVM implementation on the smartcard use the stack for all arguments or just have a single global memory array, we might save some memory by casting before calling the method. Secondly, purely speculatively, if Java, was lazy similar to functional programming languages, it might see that the desired value is a `short` and compute it as such instead of computing it as an `int` and casting it after method invocation.

- In some methods, such as `pubkToBytes`, we generate new `byte[]`s when they are needed. The method then returns a reference to this allocated memory. This design decision is different from conventions in other programming languages. We might, for example, switch to a C-style `byte[]` management, where `pubktoBytes` takes a reference to a destination `byte[]` as an additional argument, instead of creating it itself. This doesn't change anything in the implementation and does not impact memory usage, it just gives the programmer more direct control over array allocations.

- In some cases, we copy data from a message into a local `byte[]`, which isn't always necessary. If care is taken not to overwrite any important data, it might be possible to shuffle some data around on the APDU itself similar to C's `memmove` which saves on additional memory. Some simple operations, such as incrementing a value could also be done in-place, instead of turning it into a short, incrementing it and writing it back to the response APDU. This saves relatively little memory but would mean significant effort in protocol changes.
- A feature that would need to be present in a real life scenario, but isn't in our practical implementation, is the ability of the customer to choose a car he wants to rent (since different cars also have different costs and features). This is out of scope for our simulation, where you are always assigned a predetermined car.
- The database clears the auto table every time the program is started, because the program crashed if we didn't since we only have one instance of an auto, but there are multiple in the database and the database assigns a random car. In real life, this issue would not exist, it only an issue in the implementation.

8.4 GUI Tutorial

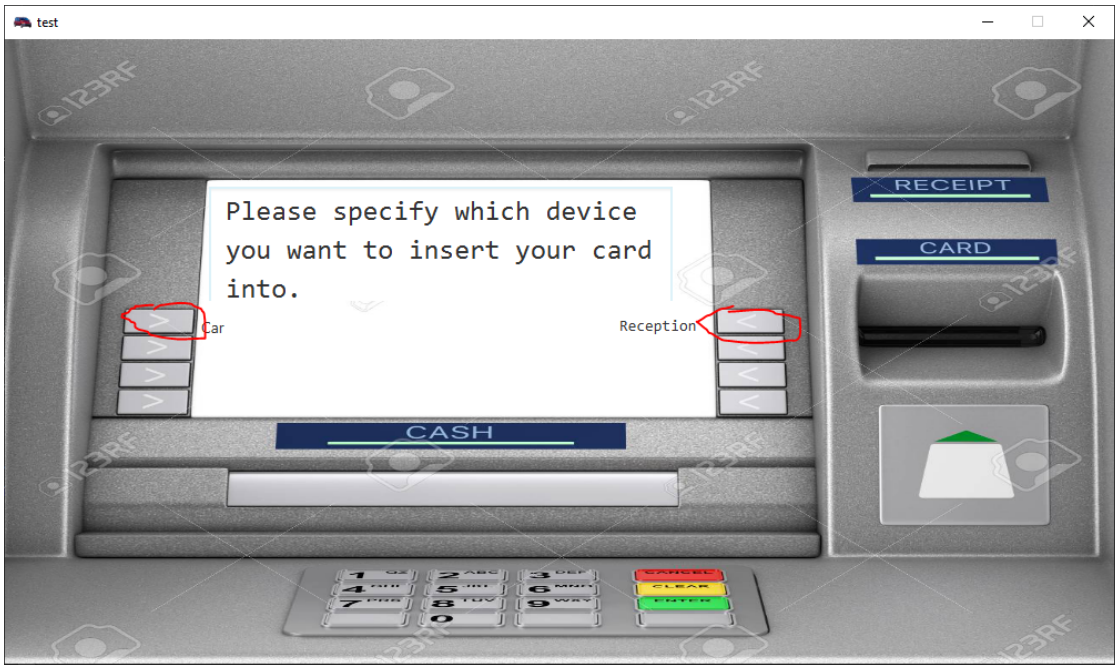
8.4.1 Home



Welcome! Please insert your card.

To properly start with the program, you need to insert your card by clicking on the area highlighted by the red circle. In practice, the cursor also indicates where you can click to perform actions.

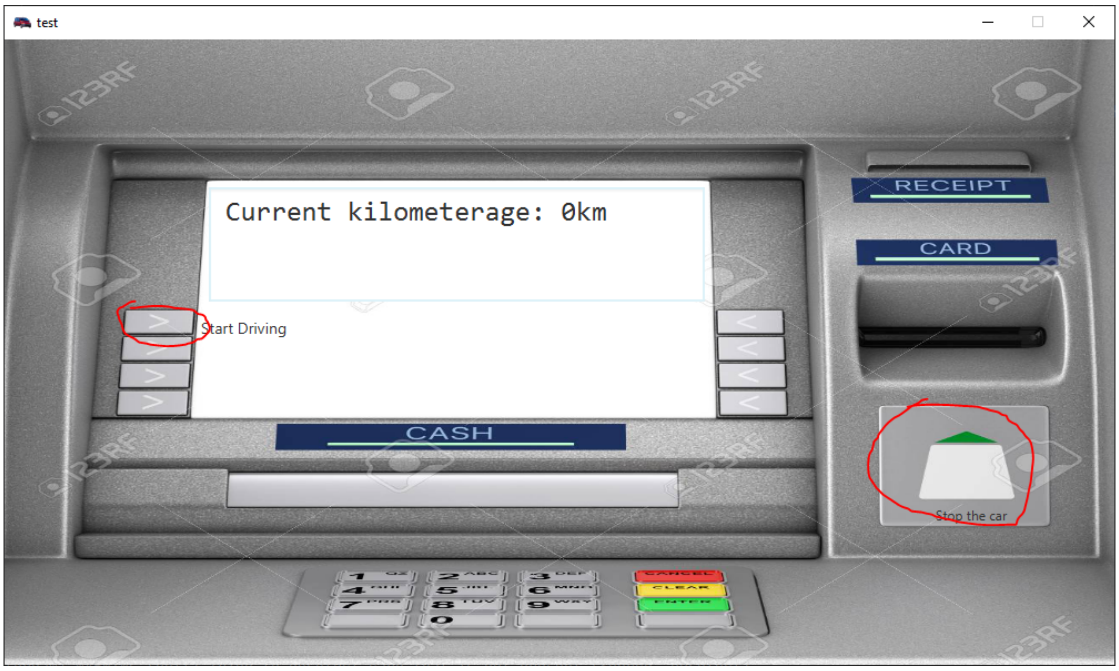
8.4.2 Startup



Please specify which device you want to insert your card into.

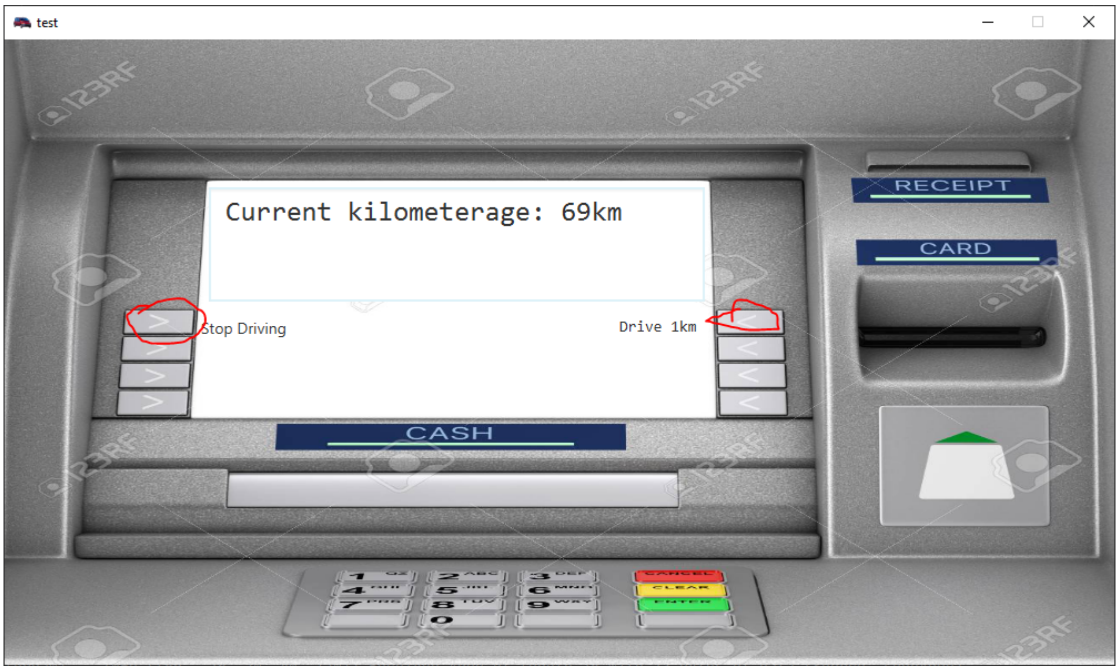
Once you insert your card, you can click on one of the highlighted buttons to indicate which terminal you would like to interact with. If you have no car assigned yet, you need to interact with the reception terminal first.

8.4.3 Car



Current kilometerage: 0km

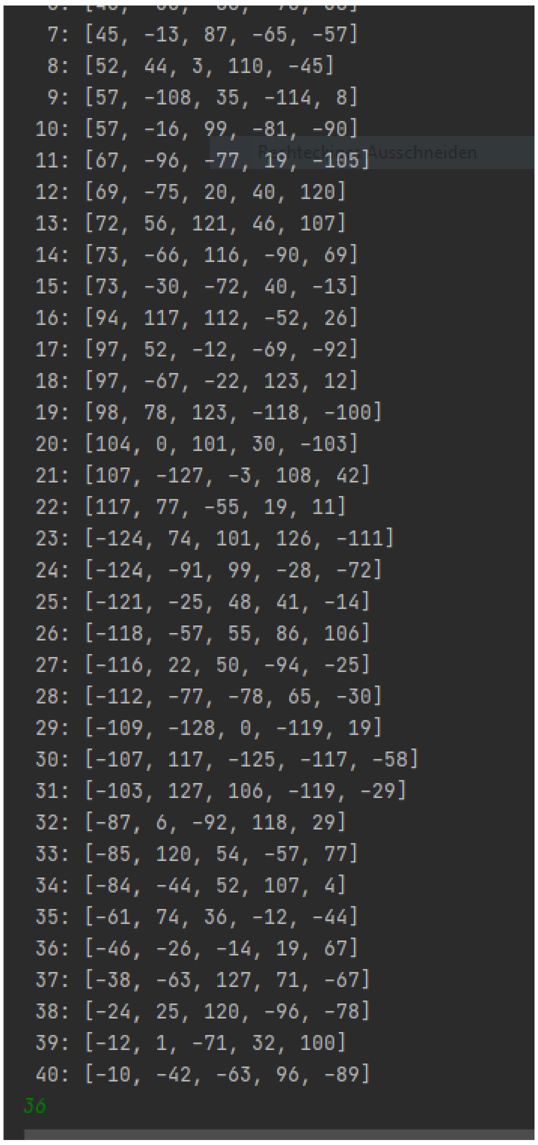
Once you choose to start a car, you are greeted by this screen. Here you can either ignite the car or leave the car.



Current kilometerage: 69km

Once you start the car you can either stop the car or drive one kilometer by clicking one of the highlighted buttons. We originally wanted the program to loop here and update the kilometerage with time, however we encountered problems with `JCSystem` and multithreading, so we opted for a button instead to still be able to simulate the protocol properly.

8.4.4 Blocking a card



```
 7: [45, -13, 87, -65, -57]
 8: [52, 44, 3, 110, -45]
 9: [57, -108, 35, -114, 8]
10: [57, -16, 99, -81, -90]
11: [67, -96, -77, 19, -105]
12: [69, -75, 20, 40, 120]
13: [72, 56, 121, 46, 107]
14: [73, -66, 116, -90, 69]
15: [73, -30, -72, 40, -13]
16: [94, 117, 112, -52, 26]
17: [97, 52, -12, -69, -92]
18: [97, -67, -22, 123, 12]
19: [98, 78, 123, -118, -100]
20: [104, 0, 101, 30, -103]
21: [107, -127, -3, 108, 42]
22: [117, 77, -55, 19, 11]
23: [-124, 74, 101, 126, -111]
24: [-124, -91, 99, -28, -72]
25: [-121, -25, 48, 41, -14]
26: [-118, -57, 55, 86, 106]
27: [-116, 22, 50, -94, -25]
28: [-112, -77, -78, 65, -30]
29: [-109, -128, 0, -119, 19]
30: [-107, 117, -125, -117, -58]
31: [-103, 127, 106, -119, -29]
32: [-87, 6, -92, 118, 29]
33: [-85, 120, 54, -57, 77]
34: [-84, -44, 52, 107, 4]
35: [-61, 74, 36, -12, -44]
36: [-46, -26, -14, 19, 67]
37: [-38, -63, 127, 71, -67]
38: [-24, 25, 120, -96, -78]
39: [-12, 1, -71, 32, 100]
40: [-10, -42, -63, 96, -89]
36
```

When blocking a card, the console lists all card numbers in the database. Just enter the number of the card you want to block in the console, press enter and then hit the confirm button on the GUI. We modelled this using the console instead of a GUI interface, as in real life this process would be done by an employee. You can retrieve the card ID from the logs of previous interactions.