Secure mobile applications using Trusted Execution Environments

Balraj Gill (s1743204)

Minf Project(Part 1) Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

This project consists of evaluating the limitations of Trusted Execution Environments by designing and implementing an EMV compliant payment mobile application using API offered by Google's Trusty TEE.

Acknowledgements

I would like to thank my supervisor Myrto Arapinis for her guidance and support throughout the project. She forward me to resources online that were very helpful and made some processes simpler and saved alot of time.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In the past few years there has been a rapid growth in the development of mobile devices and the mobile device market. The number of people who own a mobile device is only increasing. As mobile devices continue develop and become more advanced the demand for mobile applications is rising leading to more and more people diverting to the use of mobile applications in contrast to Desktops. As you can see in daily life people can simply use their mobile device to pay in shops and make financial transactions from there device. Because of this there is a greater need to make sure that mobile applications are secure and protected from malware attacks. Protocols that involve the use of sensitive data such as passwords from a registration form or credit cards (in payment protocols) should be safe even if they run on a device that has been compromised by malicious applications or simply has low level to no security. To help us design such a protocol we will use Trusted Execution Environment (TEE).

1.2 Goals

The overall goal of the project is too evaluate the limitations of the trusted execution environment and in order to do this first we need implement a secure mobile application using the TEE. This goal can be decomposed into the following sub goals:

- Build an EMV compliant Secure Mobile Application
- Implement a server for testing the app
- Then final evaluate the application and discover the limitations

We will expand on these sub goals in the Design and Implementation chapter

Chapter 2

Background

2.1 TEE

A rich operating system (RichOS) of a device is responsible for the general functionality some examples include Android, iOS and Windows. The RichOS itself is not secure enough to run apps that involve sensitive data leading to the concept of TEE[10]. The TEE contains and operating system that runs in parallel to the RichOS , it is called the Trusted Operating system. The TEE is an isolated execution environment for code, the purpose of this is too protect that code from the rest of the unknown and vulnerable device from which threats can come from. While the RichOS handles client applications the TEE can secure the data used by those applications and provide security related services for the RichOS client application. For example, in the context of an Payment app, the RichOS will handle the user interface while the TEE will store the sensitive data like credit cards and will be responsible for cryptographic functions for encrypting transactions.

2.1.1 Trusted Applications

Any application that is run using the TEE is called a trusted Application[10].

2.1.2 Trusted User Interface

Normally the RichOS is responsible for the user interface of a Mobile Application and since the RichOS is vulnerable, it is possible for the user interface to be compromised and display false information such as a wrong money amount. The purpose of a Trusted User Interface is too allow an Application on the TEE to interact with User Directly. This ensures that the information displayed is coming directly from the Trusted Application and is the correct information. GlobalPlatform[11] is an association which standardizes the management of applications on secure chip technology. Global Platform gives a use case , relevant to this project, of Trusted User Interface. The use case is ”An example of this is if an end user makes a payment using a mobile wallet or payment application. A summary of the transaction is displayed in a new window by the TEE, ensuring that any non-secure applications stored in the rich OS environment

cannot tamper with the payment details. The end user is able to sign exactly what is shown on the screen and authenticate themselves by entering a PIN or password. ”[11]

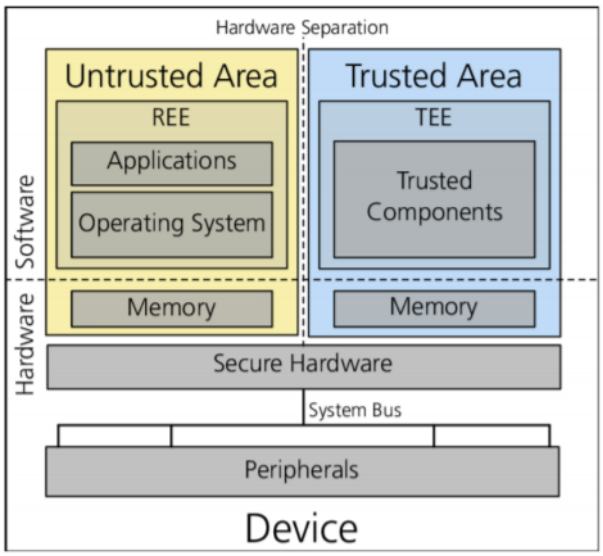

Figure 2.1: Images of a TEE and REE showing how the TEE is isolated

2.2 Trusty Android API

2.2.1 Android Keystore

The Android Keystore system [2] lets you store cryptographic keys in a container to make it more difficult to extract from the device, keys are stored using the TEE. Once keys are in the keystore, they can be used for cryptographic operations with the key material remaining non-exportable meaning the key material cannot be used outside the TEE. Moreover, it offers facilities to restrict when and how keys can be used, such as requiring user authentication for key use (e.g. entering a pin or fingerprint) or restricting keys to be used only in certain cryptographic modes. Unauthorised use of key material is prevented by enforcing apps to specify authorised use of keys. Supported devices running Android 9 (API level 28) or higher installed can have a StrongBox Keymaster. StrongBox is used to securely store keys on the Hardware security module. StrongBox also uses the TEE to test the integrity of the key.

2.2.2 Android Key attestation

Key Attestation [4] gives you more confidence that the keys you use in your app are stored in a device’s hardware-backed keystore.During key attestation, you specify the alias of a key pair. The attestation tool, in return, provides a certificate chain, which you can use to verify the properties of that key pair.

2.2.3 Android Protected Confirmation

Android Protected Confirmation [3] is another method to prevent unauthorised use of keys and to make sure it is the user that is initiating the sensitive transaction. The app will display a prompt asking the user to confirm a statement allowing the app to reaffirm that it is the user that is initiating the sensitive transaction. Once the user confirms the prompt, the message in the prompt is signed with a key from the Android keystore resulting in a signature. The signature means that, with high confidence, it was the user who has agreed to the prompt displayed. The key used to sign the message is stored in the TEE so it is safe from malware attacks and compromised devices. The protected confirmation is the prompt asking the user to press the power button twice to confirm the prompt after which the message is signed.

2.3 YubiKey

Sometimes a password on its own is not enough to protect data or an app because it is relatively easier to crack so we need something in addition to a password, that something is the second factor authentication. If an app supports second factor authentication than to get access to the app 2 things are required, a password and a second factor. The second factor has many forms for example an app can send a text message containing a code to enter before you can access an app. This method is more secure than a password alone but there are better second factors available. A really secure second factor is the Yubikey [5]. Yubikey is a hardware authentication device made by yubico[7]. It comes in different forms (below) in order to be used on different devices such as laptops or smartphones. The Yubikey supports authentication via One-time password, FIDO 2F2 and FIDO protocols [9]. For apps or protocols that support Yubikey all you must do is click a button on the Yubikey and that will authenticate the user and allow access to the app. Since this is on the hardware side, we can say with high confidence that it is the user who is initiating transaction by pressing the button on the Yubikey, instead of a malicious software program.

2.4 Related Work

2.4.1 Insulet

Insulet is a global leading manufacturer of tubeless patch insulin pumps (called Omni Pod Dash).This product uses a separate expensive remote control to control the Omni Pod. In 2018 Insulet demonstrated a use case for the Android protected Confirmation, the use case was to use the confirmation to confirm the value of insulin that will be injected using the pump. The purpose of this was to reduce the price of the product by eliminating a remote and make the product more convenient to use.

2.4.2 Previous Dissertations

The previous years disseration also consists of the same project except the mobile application used is different. The app they build is a login app. It uses protected

confirmation to confirm that it is indeed the user that wants to login to a sever hosted on a url.

2.4.3 Google Pay

Google pay [15] is similar to the protocol we will describe for the project. It also used tokenisation when paying. A token is generated to represent the card and when payment is made the token is used, the merchant matches the token with the card number and payment is made after the card and token are verified.

Chapter 3

EMV Compliant Mobile Payment Application

Our Mobile Application largely revolves around this protocol so in this section we will go over the protocol. This chapter will go over relevant information needed before going over the protocol, the origin of the protocol, entities involved and finally the protocol itself decomposed into 2 parts.

3.0.1 Tokenisation

Because the primary account of a card is sensitive and must be protected , payment tokens based on the primary account number used. These tokens are generated using the primary account number (PAN) and other data needed such as cryptographic nonces. In a transaction the token is used instead of the PAN, so if a transaction information is obtained by a malicious attacker the PAN is safe.

3.0.2 EMV compliance

EMV is a payment method based on a standard developed by Europay, Mastercard, and Visa. A payment method that meets these standards is said to be EMV compliant.

3.1 Protocol

3.1.1 Origin

The protocol is from a paper [6] and the paper proposes a design for the protocol and highlights the benefits of using this protocol.

3.1.2 Card Holder

The card holder is the owner of the card and is the user of the mobile app and will be the one initiating the transactions. It stores the following information:

- ID_{val}: It could either be a PIN code,or biometric fingerprint that only the user can provide

3.1.3 TEE

The trusted execution environment is responsible for storing the cryptography keys needed for encryption and decryption. Stores the following information:

- ID_{val}
- K_{ID}: cryptographic key used for token provisioning and is need when a request is sent to TSP for tokens generation
- K_{Pay}: cryptographc key used for encrypting payment tokens
- C_{CH}: counter needed to prevent replay attacks

3.1.4 Mobile Application

Responsible for the user interface where a user can confirm transactions , it stores the following information:

- TR_{ID}
- pk_{TSP}: public key used during the token requesting process to send a encrypted message to TSP requesting for token generation
- spk_{TSP}: public key for signature verification to verify tokens sent by the TSP

3.1.5 TSP(Token Service Provider)

The TSP is responsible for generating the payment tokens and verifying the validity of the tokens when making payments to a merchant. It stores the following formation:

- TR_{ID}
- K_{ID}
- K_{Pay}
- C_{TSP}: CPay is the counter value of the latest token validated by the TSP for a payment. If the token is too old, the payment will be refused.
- C_{Tok}
- C_{Pay}
- sk_{TSP}: private key used to decrypt messages encrypted by pk_{TSP}
- ssk_{TSP}: private signature key used to sign messages, the signed messages are verified using spk_{TSP}

3.1.6 Merchant/Point of Sale

Merchant that initiates the transaction. It stores the following information:

- M_{ID}: the unique identity of the merchant could be a number of a string
- spk_{TSP}: public key for signature verification to verify tokens sent by the Mobile Application

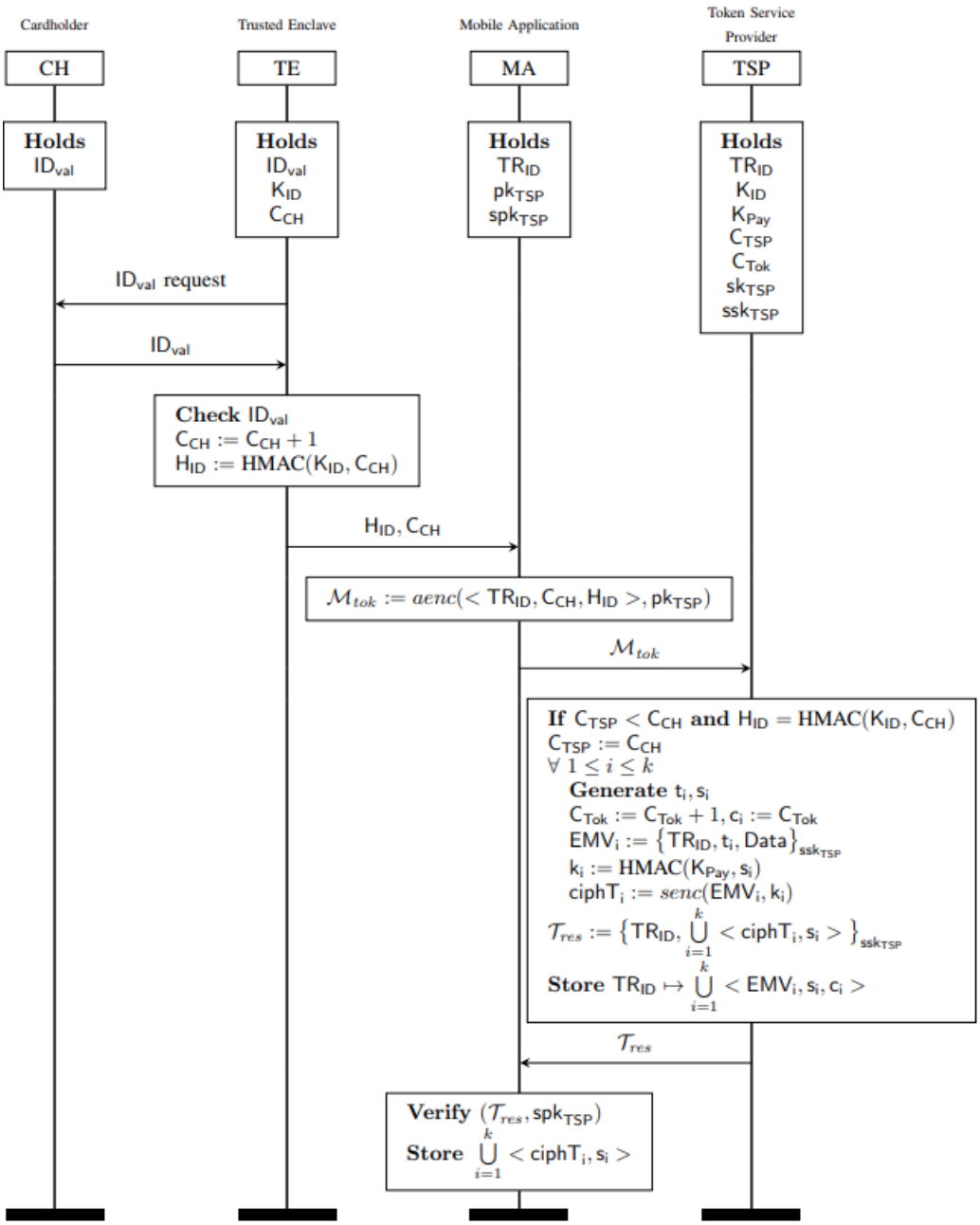

Figure 3.1: A Token Generation process, design taken from paper [6]

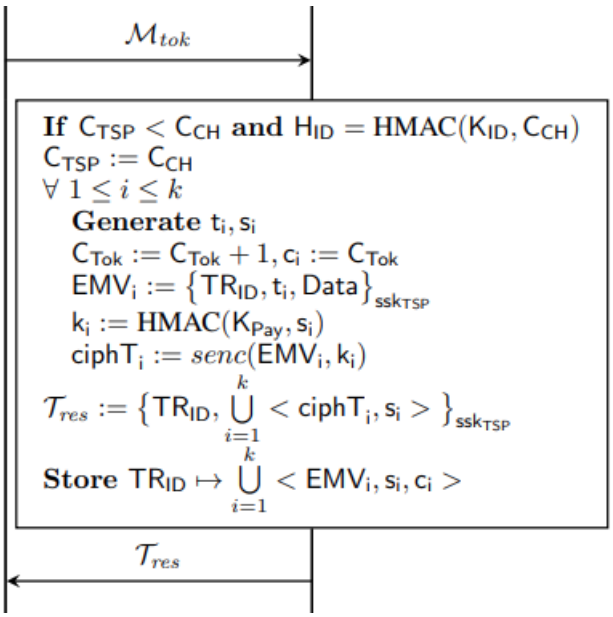

Figure 3.2: Token generation pseudo code

3.2 Token Generation

The proposed design in the paper can be decomposed into two parts with the first part being token generation/retrieval and the second part being the actual usage of the tokens via a merchant. Figure 3.1 shows the token generation process.

3.2.1 Request Tokens

This process starts with the TEE requesting the ID_{val} from card holder meaning the card holder has to enter a pin or use fingerprint biometric. Then if the pin is correct C_{CH} is incremented by one and a hash-based message authentication code (HMAC) H_{ID} is generated using the K_{ID} key. Then a token request M_{tok} is created which is just the the concatenation of TR_{ID}, C_{CH} and H_{ID} encrypted using the public key of TSP pk_{TSP}. This token request is sent to the TSP to be processed, this request is created in the Mobile App.

3.2.2 Generating Tokens

Once the TSP gets the token request it unpacks the values in it and computes its own H_{ID} using the C_{CH} received from the token request and computing the HMAC using the K_{ID} it also has. If the computed H_{ID} is equal to the one received from the token request than TSP knows the C_{CH} hasn't been tempered with since it was sent from the Mobile App. In addition the TSP also checks if the counter it has C_{TSP} is less than the counter C_{CH} in the token request. If the received counter from token request is less than the counter TSP has, then this indicates an unauthorised use of a token. Else the TSP proceeds to generate tokens.

In response to the token request the TSP sends back a response T_{res}. Firstly the TSP

updates its own counter by setting its C_{TSP} to the value of C_{CH}. Then depending on the amount of tokens generated a loop is used. For example if TSP decides to make 5 payment tokens each time this would be the process:

- Firstly 5 nonces s_i would be generated along with 5 tokens , these are associated with each EMV packet and will be used to generate and encrypt the packets in the next steps. t_i is just a randomly generated string of a particular size to ensure each payment token is different. T_{res} initially just contains the TR_{ID}.
- Then an EMV packet EMV_i is generated and this is just the concatenation of TR_{ID}, a token and Data (this is any extra information needed to generate a token such as card number , expiry date and cvv 3 digit number)
- The EMV packets above needs to be symmetrically encrypted so an encryption key k_i, unique to each packet, is generated. The key is just the HMAC of a nonce s_i using the K_{Pay}.
- Then we encrypt each EMV packet with a k_i generated above. The final step is to append to T_{res} each emv packet generated and its corresponding nonce which was used to generate its encryption key. So for 2 tokens T_{res} would be (TR_{ID},EMV_1,s_1,EMV_2,s_1)
- The Tsp also stores a mapping of EMV packet to a counter value c_i , this value is usefull for when we spend tokens. If the packet we just spent has counter value 5 then all the tokens with counter value less than 5 get deleted even if they have not been used. This helps prevent replay attacks.
- at the end T_{res} gets sent back to the Mobile App.

3.3 Spending the Tokens

Spending the tokens has the following steps:

- Firstly the merchant initiated a payment by sending the Mobile application its ID and a price, the mobile app forwards this info ,in addition a nonce s for the oldest remaining EMV packet, to the TEE
- Then the TEE asks the user to check the price and authorise the payment, if authorised the TEE generates two things. Firstly an encryption key k to decrypt an EMV packet (generated by computing the HMAC of nonce s using K_{Pay}). Secondly a message T_{val} which is an message authentication code computed by encrypting the concatenation of M_{ID},price and nonce s with K_{Pay}. These two values get sent to the Mobile application.
- The mobile app then decrypts the oldest remaining EMV packets using the key s received from TEE and verifies the packet using the public signing key spk_{TSP}. The signing key , the decrypted EMV packet and T_{val} gets sent to the Merchant.
- the merchant also verifies the EMV packet using the public signing key, and then sends the EMV packet, T_{val}, Merchant id M_{ID} and the price to the TSP.

- The TSP retrieves the EMV packet , checks if the EMV packet is unused and also computes a Message authentication code by encrypting the concatenation of M_{ID},price and nonce s with K_{Pay}. If this computed value is equal to the T_{val} received then we know that the price and merchant id displayed on mobile application is the same as id the merchant sent to the TSP. Then the TSP proceeds to delete the EMV token and confirm the payment by sending notifications.

3.4 Problems with implementation

We go over the protocol from the paper in the previous sections but in this section we will highlight problems with the proposed protocol.

- One major problem is that we do not have access to the TEE of our Mobile APP. So we cannot store our counters on the TEE. Only way to access the TEE is to use the API that is specifically used to communicate with the TEE.
- ID_{val} cannot be stored on the TEE so we cannot use it when we need user authorisation for a transaction.
- The Paper[6] assumes that both TSP and TEE already have keys K_{ID} and K_{Pay}. Symmetric Keys cannot be extracted from or put into the Android Keystore so the encryption keys K_{ID} and K_{Pay} cannot be shared with the TSP and vice versa.

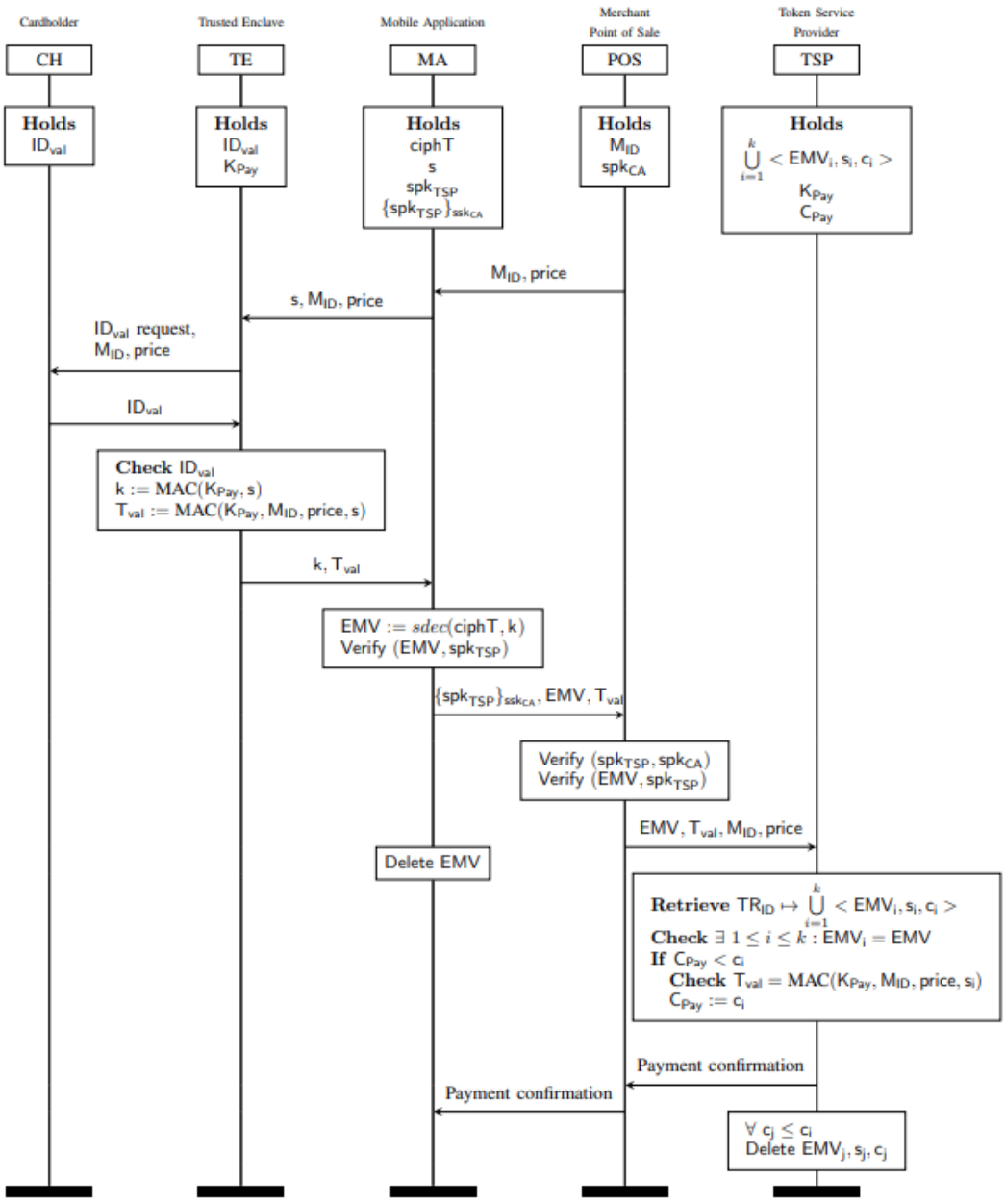

Figure 3.3: Token generation pseudo code

Chapter 4

Design and Implementation

This chapter discusses the design of the finalised protocol, with changes discussed in previous chapter, and goes over the implementation details such as what libraries were used and any challenges faced. The corresponding code for the protocol will be placed in Appendix , since there are around 800 lines of code.

4.1 Changes to protocol

- For the purpose of this project we can store the counter C_{CH} on the Mobile Application itself. For storage we used Android Shared Preferences[9].The reason for this is because the TSP also check the value C_{CH} before generating tokens, so if the value has been tempered with, tokens wont be generated and sent to the mobile app.
-
- The following can be done to deal with problem of being unable to import/export keys from TEE First generate 2 Symmetric Keys in the TEE (KeyA and KeyB), then generate 2 keys (K_{ID} and K_{Pay}) normally and hence they can be accessed. Then we immediately encrypt K_{ID} with KeyA and K_{Pay} with KeyB. Store these encrypted keys in the APP and whenever the keys are needed, decrypt them using the keys in the TEE. After this we concatenate the generated keys K_{ID} and encrypt them with the public key pk_{TSP} and send them to the TSP. The TSP will decrypt,extract and store the keys received. So now both TSP and TEE have copies of symmetric keys. Immediately after we set any variables used for the keys to null in order to decrease the time keys spend in memory.
- To replace the ID_{val} for user authentication, Android protected Confirmation is used [3]. Instead of asking for a pin or bio metrics , the user is shown a prompt to confirm the message shown on the prompt. For example when generating token the prompt tells the user that more tokens need to be generate and asks for permission.
- When spending tokens Another protected confirmation is shown to user asking

to confirm the message showing merchant id and price , this message is signed and sent to TSP. After the TSP receives this signed message it will use the Id and price received from merchant to verify the signed message. Only when the message is verified does the tokens get spent. This another mechanism built to ensure merchant does not change price but this uses the TEE.

4.2 Design of protocol

4.2.1 Assumptions

Even though some changes were made to the original protocol the assumptions raised in the paper[6] are still relevant to this project. We assume the following is true:

- the TSP is assumed to be not malicious and the operations are carried out as needed by the Mobile Application
- the TEE is assumed to be honest meaning it only processes requests when the user is authenticated.
- There are no assumptions about the Mobile Application in terms of security because our protocol is designed in such a way that even if the mobile application is compromised , it will not allow unauthorised usage of payment tokens.
- we assume the app will be run on an android phone with API Android 9 (API level 28) or higher as Android Protected Confirmation[3] only works on this android version or higher
- we assume the signing public key issued by the TSP such as spk_{TSP} are correct and so do not need to be verified by a third party Certification Authority.
- We cannot assume anything about the merchant, it may or may not be malicious. The protocol minimises the impact a malicious merchant can have for example if a merchant cancels a payment and has the Emv packet, it can only be used once and for the amount agreed upon. Secondly the E is valid until another EMV packet is used. This is bacause when new token is used,all the older tokens get deleted regardless of whether that are unused or not.

4.3 Libraries

For the communication between the mobile app and server we use Volley[14]. Volley is an HTTP library that makes networking for Android apps easier and most importantly, faster.

4.3.1 Mobile Application

The mobile application is written in Java using the Android Studio. Android Studio allows you to create an emulator to run your app. You can choose which version of android the emulator uses. The graphical user interface is quite simple as it just

Figure 4.1: Main Screen

consists of one button called Start Payment and a text box to display relevant info such as cancelled payment.

4.3.2 Servers

TSP and Merchant operations essentially consist of receiving a message, performing some operations using the message and finally replying back or forwarding a message. Taking into account the previous point we can emulate the TSP and Merchant as web servers The TSP and the merchant entities will be implemented as servers in Python. Python Flask[8] is used to build the servers , Flask is used because it is lightweight and sufficient enough for this project. Flask allows the the server to be restarted quickly when changes are made , making the Debugging process simpler and faster. You can define routes in flask server and they act as url links. The server is hosted on local host, http://192.168.0.12:5000/. And we can append a particilar string to it go to a specific page. For example when we send a token request from Mobile we send it to http://192.168.0.12:5000/tokenRequest. And the python server code in figure 4.4 below is triggered. The merchant and TSP will coded in the same file and they will be seperated by routes. So when the mobile app needs to send a message to the merchant it can do so by using http://192.168.0.12/merchant.

4.4 Token Generator

In this section we will go our the implementation of First part of the protocol, the Token generation. The design is the same as described in chapter 3 except now the changes mentioned in 4.1 have been applied. When a user clicks the payment button, the app checks if it has any EMV packets left, and depending on this one of the images in figure 4.3 (below) is shown. If there are packets left then the prompt on the left in

figure 4.3 is shown, if there are no tokens then the prompt on the right is shown asking user's permission to confirm the prompt. If the right prompt is shown then tokens are generated after the user confirms prompt.

4.5 Token Spending

If there are EMV packets remaining then user is shown the left prompt in figure 4.3, else the token generation described above is triggered. Once the user has confirmed the merchant price, the oldest token stored is obtained from the APP (it is also supposed to be verified here but that part was not implemented). We ask the TEE to decrypt the K_{Pay} key so we can use it to decrypt a EMV packet. And the rest of the steps are the same as described in Section 3.3. After the packet is sent to merchant and finally the TSP. If the packets correct and the merchant info sent to mobile and and TSP are equal the payment is confirmed and the packet is deleted on TSP as well.

4.6 Challenges

The servers were built in python and Mobile app in Java and a large part of the project consisted of encryption in java, decryption in python and vice versa. This raised a lot of issues when debugging as it was difficult to pin point the source of the error when decrypting EMV packets as both the server and app needed to be checked. When we encrypt something it returns a bytearray and this needs to be encoded to String before we send it to Mobile app or TSP server. A lot of issues were raised because trying to match the encoding on 2 different entities.

Because of the issues mentioned above combined with the time constraints the token verification parts were not implemented. Android protected confirmation was only partially implemented, the public key was sent to the TSP,the prompt was displayed and the signed message was also sent. But the signature verification was not implemented.

A significant time was spent trying to figure out and implement the changes needed for the protocol.

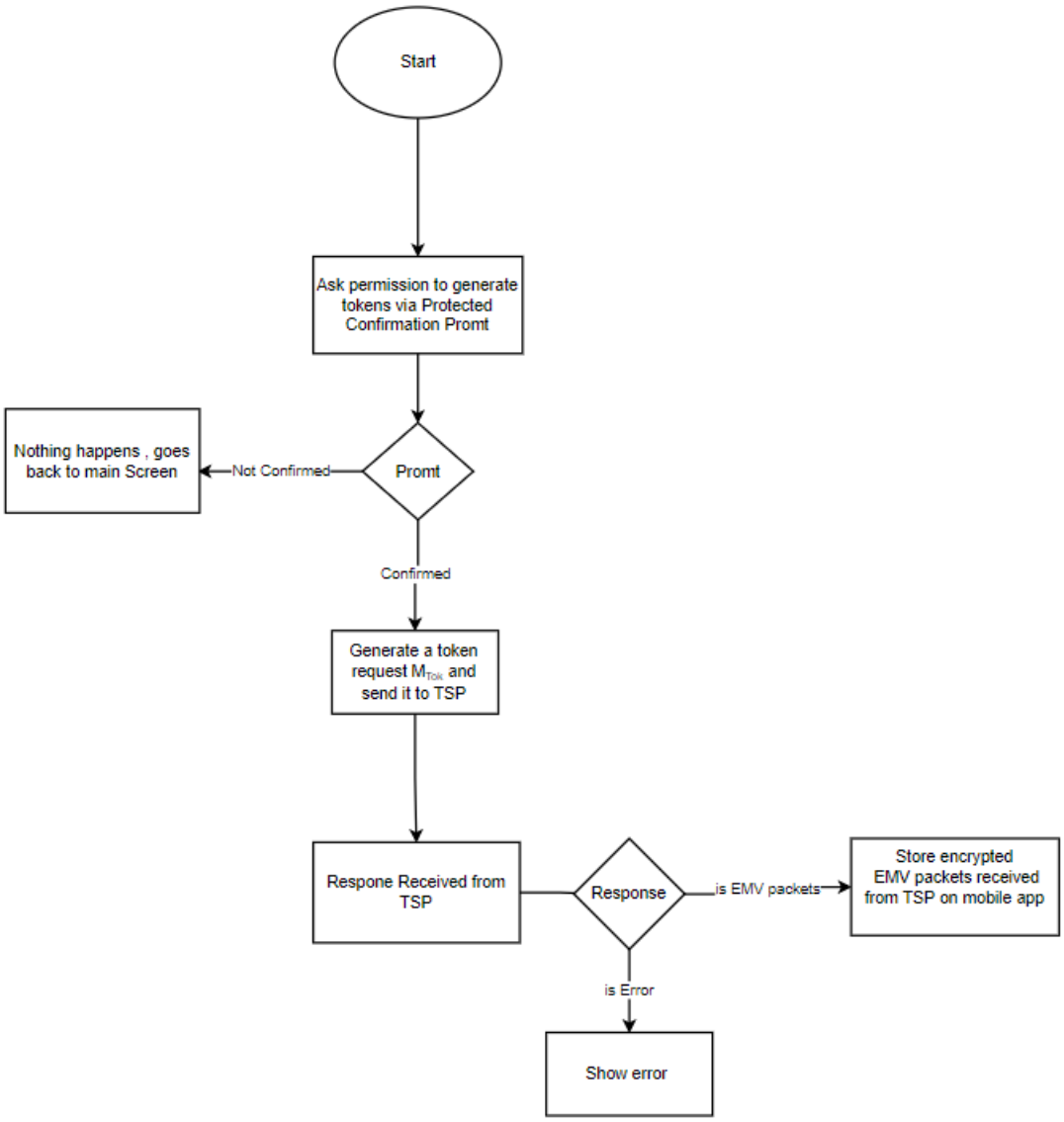

Figure 4.2: Flow chart describing the flow of programming logic for generating tokens

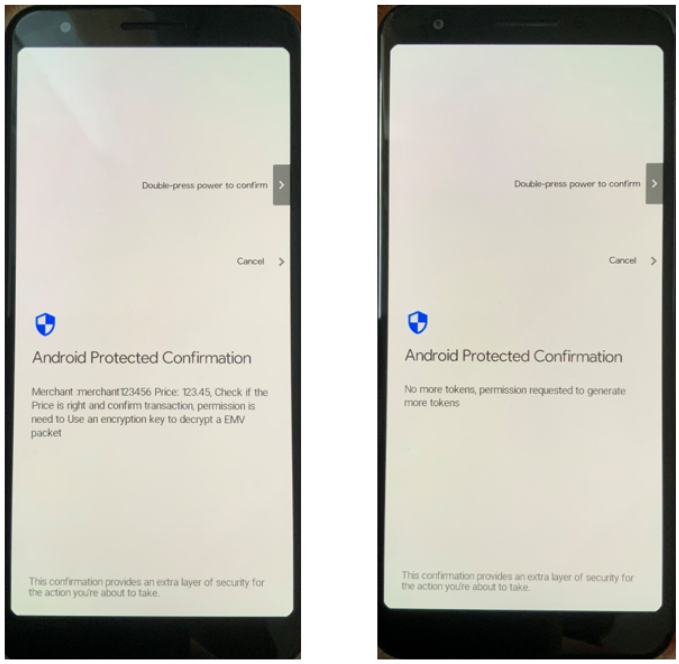

Figure 4.3: Flow chart describing the flow of programming logic

```python
@app.route('/tokenRequest', methods=['GET','POST'])
def get_data():

    msg = request.form['msg']
    cipher = PKCS1_v1_5.new(key)
    plainText = (cipher.decrypt(b64decode(msg), "Error decrypting the input string!"))
    m_tok = (b64encode(plainText)).decode("utf-8").split("separate")
    tr_id = m_tok[0]
    counter_ch = int(m_tok[1])
    H_id = m_tok[2]
    global C_TSP,C_TOK,Emvtokens

    signature = hmac.new(urlsafe_b64decode(Kid), str(counter_ch).encode(),hashlib.sha256)
    b64 = b64encode(bytes.fromhex(signature.hexdigest())).decode()  #Mtok encrypted to check is hid received = this
    T_res = tr_id

    if (H_id == b64 and C_TSP <= counter_ch):
        print("token request is valid")             #if token request is valid
        C_TSP = counter_ch
        tokens_i = gentokens(5)                     # t_i
        s_i_nonce = gen_nonces(5)   ## 18 zeros     # nonce s_i
        T_res =  tr_id + "splithere"

        for index in range(3):                    #loop for generating 3 tokens

            C_TOK +=1
            C_i = C_TOK

            key_i_raw = hmac.new(Kpay, s_i_nonce[index].encode(),hashlib.sha256)
            key_i = (b64encode(bytes.fromhex(key_i_raw.hexdigest())).decode()).encode()

            EMVi = tr_id + tokens_i[index]
            iv = s_i_nonce[index][0:16]                                    #Initialization vector needed for encryption algoritm AES

            cipher2 = AES.new(key_i[0:16], AES.MODE_CBC, iv.encode())
            ciphT_i = cipher2.encrypt(pad(EMVi.encode(),16))         #encrupt the EMV packet using Kid

            T_res += str(urlsafe_b64encode(ciphT_i)) + "nonceSeparation" + s_i_nonce[index] + "nonceSeparation"    #append the token to T_res
            Emvtokens[EMVi] = C_i      #store the tokens as dictionaries which key being a token and value being counter c_i

    else:

        print("counters did not match or Hid did not match computed hid")
        T_res = "Error in generating tokens, pls try again"

    return T_res
```

Figure 4.4: Python code for generating tokens

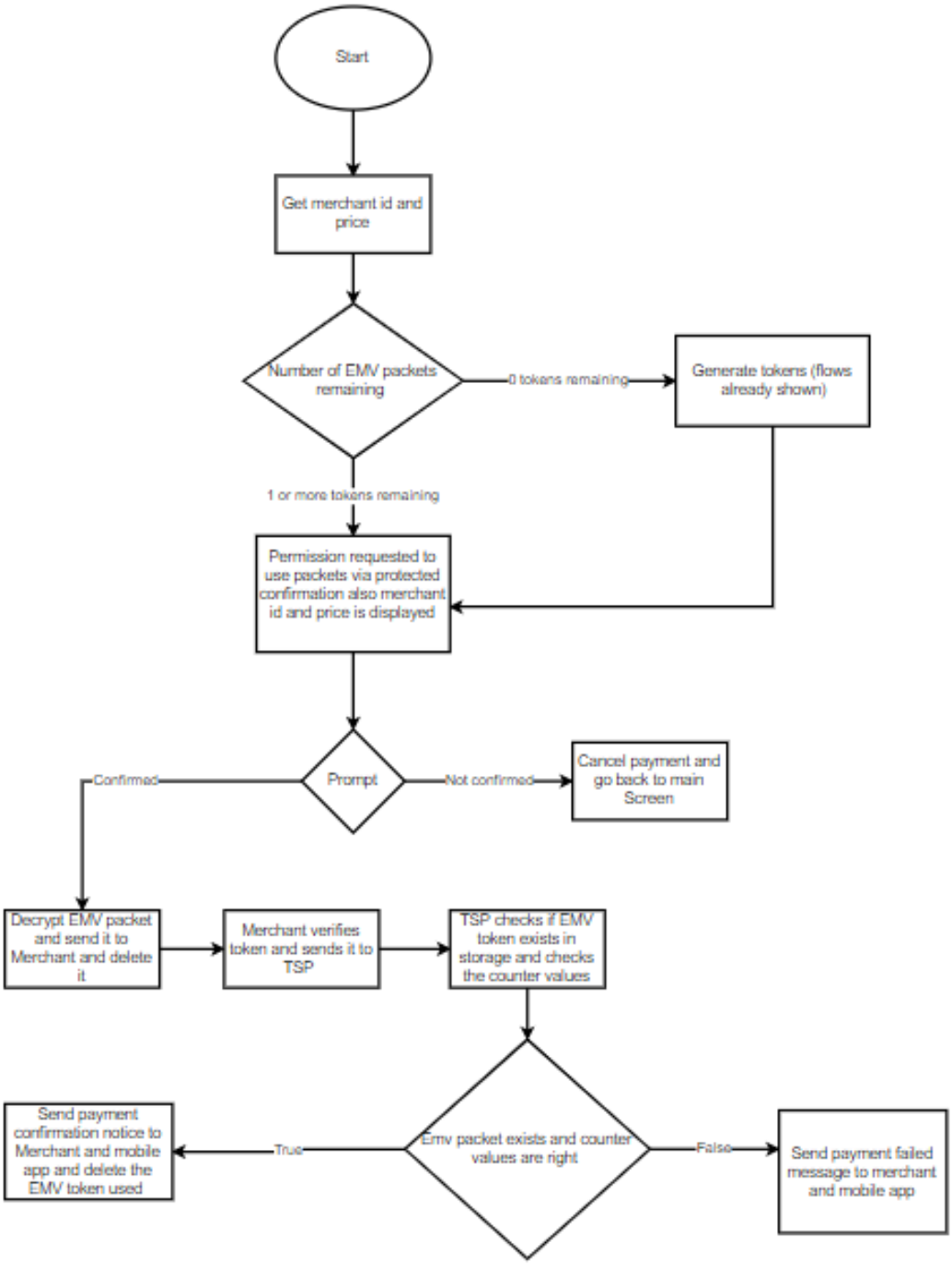

Figure 4.5: Flow chart describing how tokens are spent

Chapter 5

Evaluation

In this chapter we will evaluate the App based on the security services/functions it provides. The usability is aspect of the app is not evaluated because there is not much to evaluate, the app is simple to use.

5.1 Preliminary Evaluation and Testing

The basic functionality of the Application was tested by running it on a Google Pixel 3 mobile phone. Because simulating actual attacks is time consuming and left for future work we made a small workaround for the tests.To simulate an attack in the case where a malicious user has root access and can manipulate the variables, we simply make small changes to the TSP server code.First line of the python code we change the inequality C-TSP ¡= counter-ch to "¿". This will means that the counter received from Mobile App is lower than the one on the TSP and the mobile app requested to generate EMV packets when it still had 1 or more packets remaining. In this case the TSP simply returns an error message and does not generate any more tokens.

5.2 Mobile App Security Checklist

OWASP[12] is a non-profit foundation whose vision is **"Define the industry standard for mobile application security."** and there purpose is to write **" a security standard for mobile apps and a comprehensive testing guide that covers the processes, techniques, and tools used during a mobile app security test, as well as an exhaustive set of test cases that enables testers to deliver consistent and complete results."** OWASP defines Mobile Application Security Verification Standards (MASVS) which is just a check list of requirements that indicate how secure a Mobile Application is. The checklist is categorised based on different aspects of a mobile application. The categories are as follow:

- V1 Architecture, design and threat modelling
- V2 Data Storage and Privacy

- V3 Cryptography
- V4 Authentication and Session Management
- V5 Network Communication
- V6 Platform Interaction
- V7 Code Quality and Build Settings

OWASP also provided checklists for resiliency against reverse engineering but this check list is out of scope for this project, the categories mentioned above are sufficient to evaluate the Mobile Application. The Mobile Application Security Verification Standard also has levels, the level depends on how many checklist an app passes in a particular category, for example if an app passes all tests in Cryptography category than the app is considered ASVS level 2 in Cryptography.

Figures 5.1, 5.2, 5.3 above shows all the checklists the app passes in each of the 7 category highlighted previously. If an App passes all checklists that have a blue tick, then the app is considered to be be ASVS Level 1. To be considered Level too the app has to pass all the checklists. N/A in the figures above mean that the app does not implement the service/function described in that checklist.

As you can see from the figures above the app passes almost all of the checklists in Level 2. There is only one aspect the application does not pass in and that is Network Communication 5.1. In the context of our application , this means that when the mobile application receives an EMV packet from the TSP it does not verify the EMV packet is from the TSP using public key pk_{TSP}.

The mobile application is given a MASVS compliance score which is based on the average of the percent of applicable checklists passed in each category, the possible scores are 1,2,3,4 and 5 with 5 being the best score. The table percentage of applicable tests passed is below.

As you can see from figure 5.4 the app pass all applicable tests apart from the one in V5. Based on the table above the average is 675 percent/ 800 percent = 0.84375. The compliance score is 5x0.84375 = 4.21875 = 4 (rounded to 4). A score of 5 means that an app is completely safe based on the checklists provided by OWASP.

5.3 Advantages

Root access grants a user access to main storage and any keys stored in main storage can be compromised and the malicious attacked can use these keys. But the hardware backed keystore keys are stored separately from the main storage and hence cannot be accessed by a root access user. Keys cannot be extracted from the Keystore.

ID	MSTG-ID	Detailed Verification Requirement	Level 1	Level 2	Status	
V1		**Architecture, design and threat modelling**				
1.1	MSTG-ARCH-1	All app components are identified and known to be needed.	✓	✓	N/A	
1.2	MSTG-ARCH-2	Security controls are never enforced only on the client side, but on the respective remote endpoints.	✓	✓	N/A	N/A because we assume the TSP is safe
1.3	MSTG-ARCH-3	A high-level architecture for the mobile app and all connected remote services has been defined and security has been addressed in that architecture.	✓	✓	Pass	Yes we list the entities involved and explain their services such as the TEE
1.4	MSTG-ARCH-4	Data considered sensitive in the context of the mobile app is clearly identified.	✓	✓	Pass	
1.5	MSTG-ARCH-5	All app components are defined in terms of the business functions and/or security functions they provide.		✓	N/A	
1.6	MSTG-ARCH-6	A threat model for the mobile app and the associated remote services has been produced that identifies potential threats and countermeasures.		✓	Pass	Threat model as been identified in disseration
1.7	MSTG-ARCH-7	All security controls have a centralized implementation.		✓	Pass	TEE controls the crytopgrahy keys
1.8	MSTG-ARCH-8	There is an explicit policy for how cryptographic keys (if any) are managed, and the lifecycle of cryptographic keys is enforced. Ideally, follow a key management standard such as NIST SP 800-57.		✓	Pass	We identify the keys used Kid and Kpay and they are stored in the TEE
1.9	MSTG-ARCH-9	A mechanism for enforcing updates of the mobile app exists.		✓	N/A	
1.10	MSTG-ARCH-10	Security is addressed within all parts of the software development lifecycle.		✓	N/A	
1.11	MSTG-ARCH-11	A responsible disclosure policy is in place and effectively applied.		✓	N/A	
1.12	MSTG-ARCH-12	The app should comply with privacy laws and regulations.	✓	✓	Pass	
V2		**Data Storage and Privacy**				
2.1	MSTG-STORAGE-1	System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.	✓	✓	Pass	TEE is used to store keys there is no other sensitive data
2.2	MSTG-STORAGE-2	No sensitive data should be stored outside of the app container or system credential storage facilities.			Pass	There is no sensitive data stored
2.3	MSTG-STORAGE-3	No sensitive data is written to application logs.	✓	✓	Pass	
2.4	MSTG-STORAGE-4	No sensitive data is shared with third parties unless it is a necessary part of the architecture.	✓	✓	Pass	
2.5	MSTG-STORAGE-5	The keyboard cache is disabled on text inputs that process sensitive data.	✓	✓	Pass	
2.6	MSTG-STORAGE-6	No sensitive data is exposed via IPC mechanisms.	✓	✓	Pass	No senstive data is shared when sharing data between functions
2.7	MSTG-STORAGE-7	No sensitive data, such as passwords or pins, is exposed through the user interface.	✓	✓	Pass	we don't use passwords in out application
2.8	MSTG-STORAGE-8	No sensitive data is included in backups generated by the mobile operating system.		✓	N/A	
2.9	MSTG-STORAGE-9	The app removes sensitive data from views when moved to the background.		✓	N/A	
2.10	MSTG-STORAGE-10	The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.		✓	Pass	Variables with data is set to null after
2.11	MSTG-STORAGE-11	The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.		✓	N/A	Was not part of the project goals
2.12	MSTG-STORAGE-12	The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.		✓	N/A	
2.13	MSTG-STORAGE-13	No sensitive data should be stored locally on the mobile device. Instead, data should be retrieved from a remote endpoint when needed and only be kept in memory.		✓	Pass	
2.14	MSTG-STORAGE-14	If sensitive data is still required to be stored locally, it should be encrypted using a key derived from hardware backed storage which requires authentication.		✓	Pass	the EMV tokens stores on APP are encrypted
2.15	MSTG-STORAGE-15	The app's local storage should be wiped after an excessive number of failed authentication attempts.		✓	N/A	

Figure 5.1: Architecture, design and threat modelling + Data Storage and Privacy

V3		**Cryptography**					
3.1	MSTG-CRYPTO-1	The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.	✓	✓	Pass	Keys stored on TEE, no hardcoded keys accept public keys
3.2	MSTG-CRYPTO-2	The app uses proven implementations of cryptographic primitives.	✓	✓	Pass	AES algorithm used with CBC mode, meets the standards
3.3	MSTG-CRYPTO-3	The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.	✓	✓	Pass	Best modes are used for Encryption algorothms
3.4	MSTG-CRYPTO-4	The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.	✓	✓	Pass	
3.5	MSTG-CRYPTO-5	The app doesn't re-use the same cryptographic key for multiple purposes.	✓	✓	Pass	Kid and Kpay used to encrypt or decrypt tokens only
3.6	MSTG-CRYPTO-6	All random values are generated using a sufficiently secure random number generator.	✓	✓	Pass	
V4		**Authentication and Session Management**				
4.1	MSTG-AUTH-1	If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.	✓	✓	N/A	Again we assume the TSP is safe and not malicious
4.2	MSTG-AUTH-2	If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user's credentials.	✓	✓	N/A	
4.3	MSTG-AUTH-3	If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm.	✓	✓	N/A	
4.4	MSTG-AUTH-4	The remote endpoint terminates the existing session when the user logs out.			N/A	
4.5	MSTG-AUTH-5	A password policy exists and is enforced at the remote endpoint.	✓	✓	N/A	
4.6	MSTG-AUTH-6	The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.	✓	✓	N/A	
4.7	MSTG-AUTH-7	Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire.	✓	✓	N/A	
4.8	MSTG-AUTH-8	Biometric authentication, if any, is not event-bound (i.e. using an API that simply returns "true" or "false"). Instead, it is based on unlocking the keychain/keystore.		✓	N/A	
4.9	MSTG-AUTH-9	A second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced.	✓	✓	N/A	has not been implmented
4.10	MSTG-AUTH-10	Sensitive transactions require step-up authentication.	✓	✓	N/A	
4.11	MSTG-AUTH-11	The app informs the user of all sensitive activities with their account. Users are able to view a list of devices, view contextual information (IP address, location, etc.), and to block specific devices.		✓	N/A	
4.12	MSTG-AUTH-12	Authorization models should be defined and enforced at the remote endpoint.	✓	✓	N/A	
V5		**Network Communication**				
5.1	MSTG-NETWORK-1	Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.	✓	✓	Fail	Was not able to implement token verification
5.2	MSTG-NETWORK-2	The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.	✓	✓		
5.3	MSTG-NETWORK-3	The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.	✓	✓	N/A	We assumed the public keys pkTSP were safe and already verified by a certification authority
5.4	MSTG-NETWORK-4	The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.		✓	N/A	We assume public key is safe
5.5	MSTG-NETWORK-5	The app doesn't rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery.	✓	✓	N/A	
5.6	MSTG-NETWORK-6	The app only depends on up-to-date connectivity and security libraries.	✓	✓	N/A	
V6		**Platform Interaction**				
6.1	MSTG-PLATFORM-1	The app only requests the minimum set of permissions necessary.	✓	✓	N/A	
6.2	MSTG-PLATFORM-2	All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.	✓	✓	N/A	
6.3	MSTG-PLATFORM-3	The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.	✓	✓		
6.4	MSTG-PLATFORM-4	The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.	✓	✓	Pass	Decrypted tokens are sent through functions
6.5	MSTG-PLATFORM-5	JavaScript is disabled in WebViews unless explicitly required.	✓	✓	N/A	No javascript is used
6.6	MSTG-PLATFORM-6	WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.	✓	✓	N/A	
6.7	MSTG-PLATFORM-7	If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.	✓	✓	N/A	
6.8	MSTG-PLATFORM-8	Object deserialization, if any, is implemented using safe serialization APIs.	✓	✓		
6.9	MSTG-PLATFORM-9	The app protects itself against screen overlay attacks. (Android only)		✓	N/A	
6.10	MSTG-PLATFORM-10	A WebView's cache, storage, and loaded resources (JavaScript, etc.) should be cleared before the WebView is destroyed.		✓	N/A	
6.11	MSTG-PLATFORM-11	Verify that the app prevents usage of custom third-party keyboards whenever sensitive data is entered.		✓	N/A	

Figure 5.2: Cryptography + Authentication and Session Management + Network Communication + Platform Interaction

V7		Code Quality and Build Settings				
7.1	MSTG-CODE-1	The app is signed and provisioned with a valid certificate, of which the private key is properly protected.	✓	✓	N/A	App has not been published on appstore so not needed
7.2	MSTG-CODE-2	The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).	✓	✓	N/A	App is a very rough prototype with small bugs
7.3	MSTG-CODE-3	Debugging symbols have been removed from native binaries.	✓	✓	N/A	
7.4	MSTG-CODE-4	Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.	✓	✓	Pass	All print statements/logging as been removed
7.5	MSTG-CODE-5	All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.	✓	✓	Pass	All libraries used are verified and secure to use
7.6	MSTG-CODE-6	The app catches and handles possible exceptions.	✓	✓	Fail	Try/Catch and throws are used to catch exceptions
7.7	MSTG-CODE-7	Error handling logic in security controls denies access by default.	✓	✓	Pass	Tokens generate/used only when user authenticates
7.8	MSTG-CODE-8	In unmanaged code, memory is allocated, freed and used securely.	✓	✓	N/A	
7.9	MSTG-CODE-9	Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.	✓	✓	N/A	out of scope for the project

Figure 5.3: Code Quality and Build Settings

	Android			
	P	F	NA	%
V1: Architecture, Design and Threat Modelling	6	0	6	100.00 %
V2: Data Storage and Privacy	10	0	5	100.00 %
V3: Cryptography Verification	6	0	0	100.00 %
V4: Authentication and Session Management	1	0	11	100.00 %
V5: Network Communication	0	1	4	0.00 %
V6: Platform Interaction	1	0	7	100.00 %
V7: Code Quality and Build Settings	3	1	5	75.00 %
V8: Resiliency Against Reverse Engineering	2	0	11	100.00 %

Figure 5.4: Table of tests passed

5.4 Limitations of TEE

The TEE's emphasis on security results in it having limited amount of functions. One limitation was already mentioned in the Problems with Implementation example. If the mobile app and another entity need to have the same symmetric keys extra steps are needed. The TSP uses a symmetric key K_{Pay} first so logically it needs to be the one to generate it, but if the TSP generates this key the TEE cannot store it. This means the keys briefly enter the vulnerable part of the memory where an attacker with root access can steal the keys.

If a malicious merchant decides to cancel a payment then the merchant will have at most 1 token that they can use with a set amount of price but this is a flaw with the token system itself and not the TEE.

A man in the middle attack is possible since we have network communication. The attack is prevented to an extent due to the messages exchanged being encrypted by algorithms that are very difficult to break. But if the attacker has access to the keys used to decrypt the EMV packets then there is not alot you can do regardless of if they are stored on the TEE or not.

The counter Cch is stored on the Mobile application and someone with a root access can change its value preventing the user from generating any more tokens, the counter could be safe if it was stored on the TEE but TEE does not allow data to be imported into it.

Chapter 6

Conclusions

We implemented an EMV compliant App using the TEE keystore , android protected confirmation. In Evaluation chapter we evaluate the App and highlight the advantages and some limitations of using the TEE.

6.1 Future Work

Due to the limited amount of time an resources some parts were left for the future and some assumptions were made. But overall the parts that were completed are sufficient for the project goals

6.1.1 Uncompleted Parts

Token verification was not implemented and needs to be implemented to achieve a full MASVS compliance score mentioned in evaluation. Android protected confirmation needs to be completed specifically the signature verification on the TSP.

6.1.2 Extra functionality

Right now the app can only pay to a merchant, we can expand on this by making it so that we can pay to another application/user.

To make the application closer to applications like apple pay, a way to represent money needs to be added to the application, currently you can infinitely generate tokens and spend them. Tokens will be used only when you have enough money.

6.1.3 Simulate Attacks and more in depth Evaluation

Do more extensive testing by simulating different types of attacks including man in the middle and an attacker with root access.

Bibliography

[1] **Google Trusty TEE**, https://source.android.com/security/trusty.

[2] **Android keystore**, https://developer.android.com/training/articles/keystore.

[3] **Android Protected Confirmation**, https://developer.android.com/training/articles/security-android-protected-confirmation

[4] **Android Key Attestation**, https://developer.android.com/training/articles/security-key-attestation

[5] **YubiKey**, https://www.yubico.com/products/

[6] 2017 IEEE European Symposium on Security and Privacy (EuroSP), **Designing and Proving an EMV-Compliant Payment Protocol for Mobile Devices**, https://ieeexplore.ieee.org/document/7961997

[7] **Trusted User Interface**, https://www.trustonic.com/technical-articles/benefits-trusted-user-interface/

[8] **Flask Server**, https://flask.palletsprojects.com/en/1.1.x/

[9] **Android SharedPreferences**

[10] **Trusted Execution Environment**, https://www.trustonic.com/technical-articles/what-is-a-trusted-execution-environment-tee/

[11] **GlobalPlatform** https://globalplatform.org/latest-news/globalplatform-publishes-specification-for-the-trusted-user-interface-on-mobile-devices/

[12] **Mobile Application Security Checklist by OWASP**, https://owasp.org/www-project-mobile-security-testing-guide/

[13] **Insulet leveraging Protected Confirmation to confirm the amount of insulin to be injected.** , https://security.googleblog.com/2018/10/android-protected-confirmation-taking.html

[14] **Android Volley documentation**, https://developer.android.com/training/volley

[15] **Google Pay by Google**, https://support.google.com/pay/merchants/answer/6345242?hl=en-GBref$_topic = 7105427$

[16] **MITM**, https://www.csoonline.com/article/3340117/what-is-a-man-in-the-middle-attack-how-mitm-attacks-work-and-how-to-prevent-them.html

[17] **Root access and Keystore**, https://www.futurelearn.com/info/courses/secure-android-app-development/0/steps/21602

Appendix A

Written Code

```
KeyGenerator keyGenerator = null;
try {
    keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,  provider: "AndroidKeyStore");
} catch (NoSuchAlgorithmException | NoSuchProviderException e) {
    e.printStackTrace();
}

final KeyGenParameterSpec keyGenParameterSpec = new KeyGenParameterSpec.Builder( keystoreAlias: "Kid",
        purposes: KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
        .setIsStrongBoxBacked(true)
        .setKeySize(256)
        .build();
try {
    assert keyGenerator != null;
    keyGenerator.init(keyGenParameterSpec);
} catch (InvalidAlgorithmParameterException e) {
    e.printStackTrace();
}

final SecretKey secretKey = keyGenerator.generateKey();
```

Figure A.1: Generating keys in the TEE

```
KeyGenerator kgen = null;
try {
    kgen = KeyGenerator.getInstance("AES");
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
kgen.init( keysize: 256);
byte[] k = "".getBytes();
byte[] encrypted = "".getBytes();
try {
    encrypted = encryptPublicWithPk(k = kgen.generateKey().getEncoded());
} catch (Exception e) {
    e.printStackTrace();
}

editor.commit();
sendtoserver(Base64.encodeToString(encrypted,Base64.URL_SAFE), serverURL: url+"/sendKeys");

byte[] ENCKeys = "didnt work".getBytes();
try {
    ENCKeys = encryptKeys(k);
    encryptedKeys = Base64.encodeToString(ENCKeys,Base64.URL_SAFE);
    k = null;

} catch (Exception e) {
    e.printStackTrace();
    Log.e( tag: "dfsdf",  msg: "exception raised");
    k = null;
}
```

Figure A.2: Generating K_{ID} and K_{Pay} and encrypting them and storing them

```
public byte[] encryptKeys(byte[] data) throws KeyStoreException, CertificateException,
    KeyStore keyStore;
    keyStore = KeyStore.getInstance("AndroidKeyStore");
    keyStore.load( param: null);
    final KeyStore.SecretKeyEntry secretKeyEntry = (KeyStore.SecretKeyEntry) keyStore
            .getEntry( alias: "Kid",  protParam: null);

    final SecretKey secretKey = secretKeyEntry.getSecretKey();

    final Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
    cipher.init(Cipher.ENCRYPT_MODE, secretKey);
    iv = cipher.getIV();
    byte[] encryption = cipher.doFinal(data);

    TextView txtHello = findViewById(R.id.textView);
    txtHello.setText(Base64.encodeToString(encryption,Base64.URL_SAFE));
    return encryption;

}
```

Figure A.3: Function used to help encrypt keys in figure above

```
public byte[] decryptKeys(byte[] data) throws NoSuchPaddingException, NoSuchAlgorithmException,
    KeyStore keyStore;
    keyStore = KeyStore.getInstance("AndroidKeyStore");
    keyStore.load( param: null);
    final KeyStore.SecretKeyEntry secretKeyEntry = (KeyStore.SecretKeyEntry) keyStore
            .getEntry( alias: "Kid",  protParam: null);

    final SecretKey secretKey = secretKeyEntry.getSecretKey();
    final Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
    final GCMParameterSpec spec = new GCMParameterSpec( tLen: 128, iv);
    cipher.init(Cipher.DECRYPT_MODE, secretKey, spec);

    return cipher.doFinal(data);
}
```

Figure A.4: Function used to decrypt data encrypted using TEE keys

```
public byte[] encryptPublicWithPk(byte[] data) throws NoSuchAlgorithmException, In

    byte[] publicBytes = Base64.decode(pkTSP_string, Base64.DEFAULT);

    X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicBytes);
    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    PublicKey pubKey = keyFactory.generatePublic(keySpec);
    Cipher cipher1 = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher1.init(Cipher.ENCRYPT_MODE, pubKey);
    final byte[] encrypted = cipher1.doFinal(data);

    return encrypted;
}
```

Figure A.5: Function used to encypt data using TSP public key

```
public String decryptToken(View view) throws NoSuchAlgorithmException, InvalidKeyException, NoSuchPaddingException,


    byte[] encoded_Kpay = decryptKeys(Base64.decode(encryptedKeys,Base64.URL_SAFE));
    String kpay = Base64.encodeToString(encoded_Kpay,Base64.URL_SAFE).substring(16,32);
    byte[] encoded_K_pay = Base64.decode(kpay,Base64.URL_SAFE);

    Mac mac = Mac.getInstance("HmacSHA256");
    mac.init(new SecretKeySpec(encoded_K_pay,  algorithm: "HmacSHA256"));
    byte[] macResult = mac.doFinal(T_res_nonce_s.get(0).getBytes());
    String key_i = Base64.encodeToString(macResult,Base64.DEFAULT);

    encoded_K_pay = null;
    kpay = null;
    encoded_K_pay = null;


    SecretKey ciphT_key_i = new SecretKeySpec(key_i.substring(0,16).getBytes(),  algorithm: "AES");
    IvParameterSpec iv = new IvParameterSpec(T_res_nonce_s.get(0).substring(0,16).getBytes());
    Cipher cipher1 = Cipher.getInstance("AES/CBC/NoPadding");
    cipher1.init(Cipher.DECRYPT_MODE, ciphT_key_i,iv);

    key_i = null;
    byte[] mtok = cipher1.doFinal(Base64.decode(T_res_ciphT.get(0),Base64.URL_SAFE));

    T_res_nonce_s.remove( index: 0);
    T_res_ciphT.remove( index: 0);

    return new String(mtok);
}
```

Figure A.6: Function used to decrypt a EMV packet

```
RequestQueue queue = Volley.newRequestQueue( context: this);
String urlTokReq = url + "/merchant";

StringRequest stringRequest = new StringRequest(Request.Method.POST, urlTokReq,
        new Response.Listener<String>() {
            @Override
            public void onResponse(String response) {
                setMerchantinfo(response);
            }
        }, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        TextView txtHello = findViewById(R.id.textView);
        txtHello.setText(error.getMessage());

    }
});

queue.add(stringRequest);
```

Figure A.7: code for getting merchant id and price

```
byte[] encoded_Kid = decryptKeys(Base64.decode(encryptedKeys,Base64.URL_SAFE));
String kid = Base64.encodeToString(encoded_Kid,Base64.URL_SAFE).substring(0,16);
byte[] encoded_K_id = Base64.decode(kid,Base64.URL_SAFE);


SecretKeySpec keySpec_hid = new SecretKeySpec(
        encoded_K_id,
         algorithm: "HmacSHA256");

Mac mac = Mac.getInstance("HmacSHA256");
mac.init(keySpec_hid);
byte[] hid_result = mac.doFinal(bytecounter);   //change this to card holder counter
String hid = Base64.encodeToString(hid_result, Base64.DEFAULT);
```

Figure A.8: Used to generate H_{ID} used in token request that gets sent to TSP

```
byte[] publicBytes = Base64.decode(pkTSP_string, Base64.DEFAULT);
String pre_mtok = trid + split + strcounter + split + hid;
byte[] testmsg = Base64.decode(pre_mtok, Base64.DEFAULT);
X509EncodedKeySpec keySpec = new X509EncodedKeySpec(publicBytes);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
PublicKey pubKey = keyFactory.generatePublic(keySpec);

Cipher cipher1 = Cipher.getInstance("RSA/ECB/PKCS1Padding");
cipher1.init(Cipher.ENCRYPT_MODE, pubKey);
final byte[] mtok = cipher1.doFinal(testmsg);

encoded_K_id = null;
kid = null;
encoded_K_id = null;
test = true;
sendtoserver(Base64.encodeToString(mtok, Base64.DEFAULT),urlTokReq);
```

Figure A.9: Tokens request M_{TOK} gets generated and send to TSP

```
public void sendtoserver(final String data, String serverURL){
    RequestQueue queue = Volley.newRequestQueue( context: this);
    StringRequest stringRequest = new StringRequest(Request.Method.POST, serverURL,
            new Response.Listener<String>() {
                @Override
                public void onResponse(String response) {
                    TextView txtHello = findViewById(R.id.textView);
                    String resp = new String(response);
                    txtHello.setText(response);
                    T_res = response;
                    if (test) {

                        String[] T_res_split = response.split( regex: "splithere");
                        String T_res_store = T_res_split[1];
                        String[] s = {};
                        s = T_res_store.split( regex: "nonceSeparation");
                        for (int i = 0; i < s.length; i++) {
                            if (i % 2 == 0) {
                                T_res_ciphT.add(s[i].substring(2, s[i].length() - 1));
                            } else {
                                T_res_nonce_s.add(s[i]);
                            }
                        }
                    }
                }
            }, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
            TextView txtHello = findViewById(R.id.textView);
            txtHello.setText(error.getMessage());
        }
    }) {
        @Override
        protected Map<String, String> getParams() {
            Map<String, String> params = new HashMap<~>();
            params.put("msg", data);
            return params;
        }

    };
    queue.add(stringRequest);
}
```

Figure A.10: function which is used to send message to TSP

```
public void testProtectedConfirm(View view) throws ConfirmationNotAvailableException, ConfirmationAlreadyPresentingExceptio
    ConfirmationPrompt confirmationPrompt = new ConfirmationPrompt.Builder( context: this)
            .setPromptText(String.format("No more tokens, permission requested to generate more tokens"))
            .setExtraData("Challenge".getBytes())
            .build();
    confirmationPrompt.presentPrompt(getMainExecutor(), createProtectedConfirm());
}

private ConfirmationCallback createProtectedConfirm() {
    return new  ConfirmationCallback() {
        View view;
        @Override
        public void onConfirmed(byte[] dataThatWasConfirmed){
            super.onConfirmed(dataThatWasConfirmed);
            Toast.makeText(getApplicationContext(), new String(dataThatWasConfirmed) , Toast.LENGTH_LONG).show();
            try {
                TextView t = findViewById(R.id.textView);
                sendTokenRequest(view);
                t.setText(String.valueOf(T_res_ciphT.size()));

            } catch (NoSuchAlgorithmException | NoSuchProviderException | InvalidAlgorithmParameterException | NoSuchPaddi
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (CertificateException e) {
                e.printStackTrace();
            } catch (UnrecoverableEntryException e) {
                e.printStackTrace();
            } catch (KeyStoreException e) {
                e.printStackTrace();
            }
        }
```

Figure A.11: Protected confirmation for Generating tokens

```
public void testProtectedConfirm2(View view) throws ConfirmationNotAvailableException, ConfirmationAlreadyPresentingExceptio
    ConfirmationPrompt confirmationPrompt = new ConfirmationPrompt.Builder( context: this)
            .setPromptText(String.format("Merchant :" + merchantid + " Price: " + price + ", Check if the Price is right and
            .setExtraData("Challenge".getBytes())
            .build();
    confirmationPrompt.presentPrompt(getMainExecutor(), createProtectedConfirm2());
}

private ConfirmationCallback createProtectedConfirm2() {
    return new  ConfirmationCallback() {
        View view;
        @Override
        public void onConfirmed(byte[] dataThatWasConfirmed){
            super.onConfirmed(dataThatWasConfirmed);
            Toast.makeText(getApplicationContext(),  text: "Prompt confirmed!", Toast.LENGTH_LONG).show();
            try {
                encrypt(view);
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
            } catch (InvalidKeyException e) {
                e.printStackTrace();
            } catch (IllegalBlockSizeException e) {
                e.printStackTrace();
```

Figure A.12: Protected confirmation for spending tokens

```python
@app.route("/genkeys")
def generatekeys():
    tsp_priv_key = RSA.generate(2048)
    tsp_pub_key = tsp_priv_key.publickey()


    with open("tsp_priv.pem", "w") as priv:
        priv.write(tsp_priv_key.export_key().decode())
    with open("tsp_pub.pem", "w") as pub:
        pub.write(tsp_pub_key.export_key().decode())

    tsp_priv_key_signature = RSA.generate(2048)
    tsp_pub_key_signature = tsp_priv_key_signature.publickey()




    with open("tsp_priv_signature.pem", "w") as priv:
        priv.write(tsp_priv_key_signature.export_key().decode())
    with open("tsp_pub_signature.pem", "w") as pub:
        pub.write(tsp_pub_key_signature.export_key().decode())

    return "success"
```

Figure A.13: Generating keys in the TSP such a pk_{TSP}