B.Tech. Project

on

Digital Payment Apps

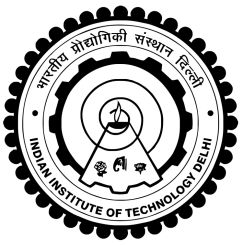
(Course Code: COD492)

Final Report - November 2024

Submitted by:-

Shreejeet Golhait - 2019CS10351
Tushman Khalse - 2019CS10411

Supervised by: Prof. Tarun Mangla



Department of Computer Science & Engineering

Indian Institute of Technology Delhi

November 2024

Acknowledgements

We would like to express our sincere gratitude to Professor Tarun Mangla, for providing us with an opportunity to work on this project as a part of our B.Tech thesis. We would also like to thank Professor Aaditeshwar Seth, Professor Rijurekha Sen and Professor Vireshwar Kumar for providing us with meaningful insights on our project that helped us guide our research direction and refine our analysis effectively.

Contents

Introduction

In recent years, India has witnessed a significant transformation in its financial ecosystem, driven by the rapid adoption of digital payment systems. Digital payment apps have become an integral part of daily transactions, offering convenience, speed, and accessibility. These applications cater to a wide range of financial activities, from paying utility bills and transferring money to shopping and booking services. The proliferation of smartphones, affordable internet, and supportive government policies like the **Digital India initiative** have fueled the growth of these platforms, making them accessible to even the remotest corners of the country.

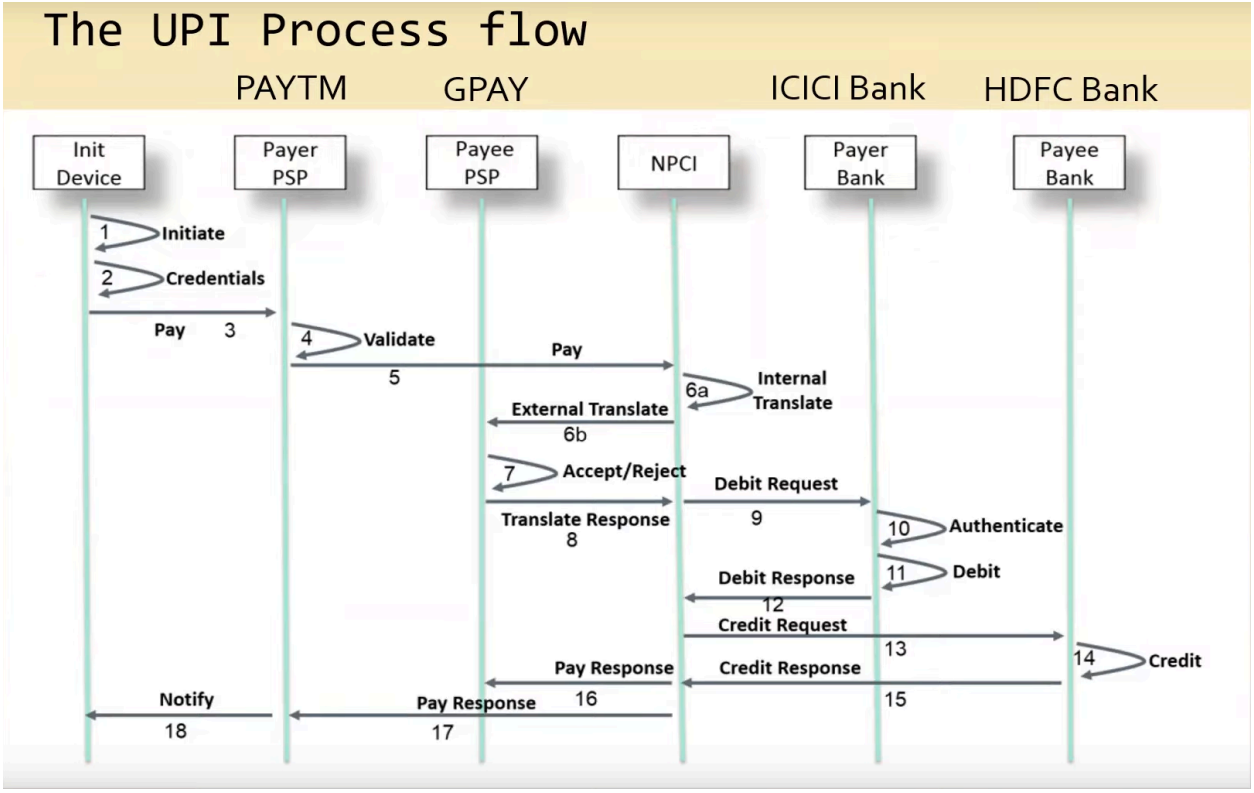
At the heart of this digital revolution lies the **Unified Payments Interface** (UPI), a groundbreaking payment system developed by the National Payments Corporation of India (NPCI). UPI has simplified digital transactions by enabling instant, seamless, and secure fund transfers between banks using mobile devices. Its interoperability across banks and apps has democratized access to financial services, ensuring that people from all socioeconomic backgrounds can participate in the digital economy. UPI has been particularly transformative for individuals without access to traditional banking services, empowering them to engage in cashless transactions and fostering greater financial inclusion.

The ubiquity of UPI has permeated every sector of the population in India today, from urban professionals and small businesses to rural farmers and street vendors. Its widespread adoption has made India a global leader in real-time digital payments, contributing significantly to the nation's journey toward a **cashless economy**. However, this rapid adoption also raises critical questions about the performance, security, and efficiency of the apps facilitating these transactions.

Given the pivotal role digital payment apps play in the country's financial landscape, studying their functionality, performance, and challenges is more important than ever. This project aims to delve into the inner workings of digital payment applications, exploring their technical underpinnings, user experience, and reliability under varying conditions. Understanding these aspects not only sheds light on how these apps contribute to India's economic growth but also provides insights into potential areas of improvement. By analyzing these platforms, we hope to contribute to the ongoing discourse on enhancing digital payment systems for a seamless and secure user experience.

Literature Review

A. How Does a UPI Transaction Work?



The Unified Payments Interface (UPI) system facilitates seamless, instant fund transfers between bank accounts using mobile applications. It operates through a structured workflow involving the user, Payment Service Providers (PSPs), banks, and the National Payments Corporation of India (NPCI), ensuring both speed and security.

The process begins when a user initiates a payment request on their app, such as Paytm or GPay. The PSP validates the request and forwards it to the recipient's PSP for verification. NPCI acts as a clearinghouse, routing the transaction between the involved banks and ensuring compliance with protocols.

When a payment is initiated, the payer's bank processes a debit request by authenticating the user and confirming the transaction. Simultaneously, the payee's bank handles the credit request, completing the fund transfer. NPCI ensures that transaction details, such as authorization and confirmations, are securely routed to all parties. Finally, both the payer and the payee receive status updates, completing the transaction lifecycle.

This multi-layered architecture ensures a robust, efficient system capable of handling high transaction volumes securely.

B. Man-in-the-Middle Attacks

A Man-in-the-Middle (MITM) attack occurs when an unauthorized entity intercepts communications between two legitimate parties. In the context of digital payment apps, this involves intercepting network traffic to view, manipulate, or steal sensitive information like payment credentials and transaction data.

MITM attacks exploit vulnerabilities in secure communication protocols by acting as a proxy between the app and the server. This can involve intercepting HTTP(S) requests, decoding encrypted traffic, or injecting malicious payloads into the communication stream.

The success of MITM attacks often depends on the target's failure to enforce strict certificate validation or implement advanced security mechanisms like certificate pinning. Such attacks highlight the need for robust cryptographic practices and app-level security protocols.

C. Certificate Unpinning

Certificate pinning is a security technique used by applications to mitigate the risks of **Man-in-the-Middle (MITM) attacks**. It ensures that an application only communicates with a server using a specific, pre-approved SSL/TLS certificate or a set of certificates. Even if an attacker intercepts the communication and provides a valid but unauthorized certificate (e.g., from a trusted Certificate Authority), the app will reject the connection because the certificate doesn't match the pinned one.

This technique is often used by high-security apps like banking or payment apps to protect sensitive data transmitted over the network.

How Certificate Pinning Works:-

1. **Certificate Trust Model:** In standard TLS connections, the server's certificate is verified against a chain of trust maintained by trusted Certificate Authorities (CAs). If the certificate is valid and issued by a trusted CA, the connection is established.

2. **Pinning a Certificate:** With certificate pinning, the application stores a copy (or hash) of the server's certificate or public key within the app itself (static pinning) or in an online server (dynamic). During a connection, the app checks if the server's certificate matches the pinned certificate. If it matches, the connection proceeds. Otherwise, it is terminated, even if the certificate is valid according to the CA.

There are two primary methods to unpin certificates: **static methods** and **dynamic methods**.

1. Static Methods of Certificate Unpinning

Static unpinning involves modifying the app itself to disable or bypass certificate pinning. This is typically done by decompiling the app, analyzing the code, and altering the relevant sections.

A. **Decompiling the APK:** Tools like **apktool** or **jadx** are used to decompile the APK and access the app's source code or bytecode.

B. **Locating the Pinning Logic:** Developers typically store certificate hashes or keys in the codebase, which are verified during the handshake process.

C. **Modifying the Pinning Code:** Bypass the certificate validation code by modifying the function that checks the pinned certificate. For example, in Android apps, this might involve overriding the checkServerTrusted() method in the TrustManager interface.

D. **Recompiling and Signing the APK:** After modifying the code, the app is recompiled and signed with a new debug or release key for execution.

2. Dynamic Methods of Certificate Unpinning

Dynamic unpinning involves modifying the app's behavior in real-time during execution, without altering the APK. This is typically done using tools like **Frida** or **Xposed Framework**. Frida is a popular tool for injecting custom scripts into a running application to bypass certificate pinning. A custom Frida script hooks into the certificate validation functions at runtime. The script can override methods like checkServerTrusted() to always accept any certificate, effectively bypassing the pinning mechanism. This works only with rooted devices.

Network Shaping

Network shaping, also known as traffic shaping or traffic control, is a process that modifies network traffic to emulate specific conditions like reduced bandwidth, increased latency, or packet loss. This enables researchers to study the behavior of networked systems under diverse real-world scenarios.

Network shaping is implemented by controlling the queuing disciplines (qdiscs) associated with network interfaces. A **qdisc** is a packet scheduler that determines how packets are queued, delayed, or dropped as they traverse a network interface. By managing qdiscs, network shaping tools like tc (Traffic Control) on Linux achieve precise control over network conditions.

A qdisc defines the rules for how packets are handled by the network interface. Qdiscs are applied to specific network interfaces (e.g., eth0, wlan0). All traffic flowing through the interface is shaped according to the configured rules.

tc is a powerful utility that allows precise traffic control through qdisc management. It works by:

a) Rate Limiting: Restricting upload or download speeds to simulate slower connections by configuring qdiscs like HTB or TBF (Token Bucket Filter).

b) Introducing Latency: Adding artificial delays to packets using Netem, simulating conditions like satellite links.

c) Simulating Packet Loss: Dropping a percentage of packets randomly or systematically to emulate unstable networks.

d) Jitter Simulation: Randomly varying packet delays to represent inconsistent network timing.

e) Packet Reordering and Duplication: Modifying the order or duplicating packets to test how systems handle corrupted streams.

Objectives

Investigation of Network Traffic in UPI Transactions

• Conduct a comprehensive analysis of network traffic generated during Unified Payments Interface (UPI) transactions to identify underlying patterns, trends, and anomalies.

• Capture and categorize packet data to understand the sequence of network calls, packet sizes, request-response times, and data flow dynamics during each phase of a transaction.

• Utilize network traffic monitoring tools to visualize and interpret the volume of traffic, concurrent connections, and potential bottlenecks in real-time transaction processing.

• Logging packet capture data of UPI transactions for various commonly used apps and figure out Server Name Indications used during the transaction.

Decryption and Data Flow Analysis in Network Calls

• Analyze encrypted payloads within UPI network packets to understand the data flow between client devices, and the payment gateways.

• Develop techniques or utilize existing tools for decrypting transaction metadata for multiple digital payment applications.

• Using decrypted metadata to isolate important network calls related to the transactions.

Development of Automation Scripts for UPI Transactions

• Design and implement automation scripts to simulate UPI transaction flows, including payment initiation, authentication, and confirmation, across various digital payment apps.

• Utilize scripting languages and automation frameworks (e.g., Python, Selenium, Appium) to automate end-to-end UPI transaction scenarios, reducing manual testing efforts and improving consistency.

• Ensure the scripts can be utilized for automating multiple transactions while recording the transaction times.

• Implement error-handling mechanisms within the scripts to log transaction failures, timeouts, and retry attempts, providing detailed insights into system performance under both normal and stressed conditions.

Cross-Comparison of UPI Apps' Network Behavior

• Simulate diverse network conditions, including high latency, limited bandwidth, and packet loss, to measure their effects on UPI transaction success rates and end-user experience.

• Conduct side-by-side comparisons of response times, error rates, and error rates across different apps.

• Analyze the correlation between degraded network conditions and transaction timeout rates, failed authentications, and retry attempts.

• Develop detailed performance metrics for each network scenario, highlighting critical thresholds where transaction success significantly diminishes.

Performance Benchmarking and Privacy Concerns

• Establish performance benchmarks for UPI applications, focusing on transaction speed, network efficiency, and error handling.

• Identifying privacy breaches done by the UPI applications using decrypted transaction data.

• Identifying security loopholes present in these apps which can be used by adversaries.

Methodology

Capturing UPI transaction network data in form of PCAP files

PCAPdroid is an Android application that captures network traffic directly from the device without requiring root access. It acts as a **VPN-based packet sniffer**, which intercepts all incoming and outgoing traffic by routing it through a virtual network interface. This app can extract PCAP files from your mobile device, which can be further analysed using software like Wireshark.

During the capture, most UPI and digital payment applications (like PayTM) use **TLS** (Transport Layer Security) for encrypting network communications. The captured data was encrypted, making it difficult to view sensitive information like API requests, payment details, and responses. Modern apps also implement **certificate pinning** and HMAC (Hash-based Message Authentication Code) checks, adding additional layers of security to prevent tampering or interception of traffic.

Server Name Indication (SNI) is an extension to the TLS (Transport Layer Security) protocol that allows a client (e.g., a mobile app or browser) to specify the hostname it is trying to connect to during the TLS handshake. This is especially useful when multiple domains are hosted on the same IP address, allowing the server to present the correct SSL/TLS certificate for the requested hostname.

Digital payment apps use HTTPS to encrypt network traffic, making it challenging to observe and analyze the data being exchanged. However, the SNI field in the TLS handshake is not encrypted (in most cases, unless Encrypted Client Hello (ECH) is implemented). Therefore, analyzing SNI provides valuable information about the backend infrastructure and servers the app communicates with.

We wrote a python script, which used the python library **dpkt**, to identify SNI present in the PCAP file. It returns all SNI it found in the PCAP file, and also plots a TLS transaction over time graph. Running the script for multiple PCAP files corresponding to UPI transactions, we were able to identify common SNIs called during transactions for those apps:

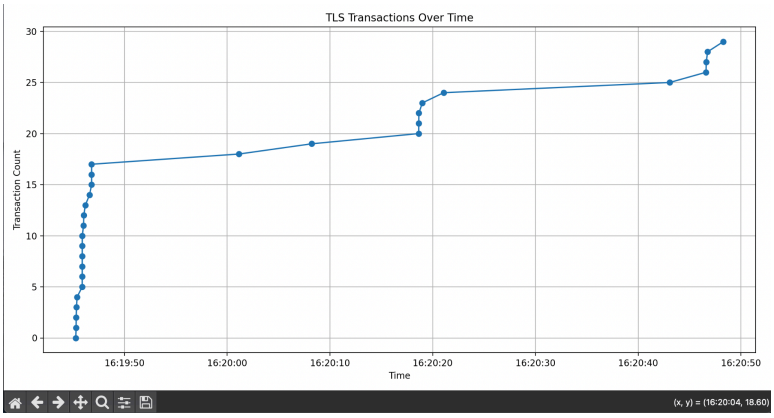
- PayTM: digitalapiproxy.paytm.com
- GPay: paymentsincentives-pa.googleapis.com, india-paisa-pa.googleapis.com
- PhonePe: apicp2.phonepe.com

```
Total packets processed: 738
IP packets found: 738
TCP packets found: 652
Possible TLS packets (port 443): 602
TLS Handshakes found: 30
ClientHello messages found: 14
SNIs extracted: 14

SNI found:
- sig.paytm.com
- graph.facebook.com
- securegw-online.paytm.in
- storefront.paytm.com
- tvybx4-launches.appsflyersdk.com
- assetscdn1.paytm.com
- ws-7cc3e8d0-d35b-4685-b603-135df578e7de.sendbird.com
- digitalapiproxy.paytm.com
- crashlyticsreports-pa.googleapis.com
- api-7cc3e8d0-d35b-4685-b603-135df578e7de.sendbird.com
- kyc.paytmbank.com

Total TLS transactions: 14
```

All SNI found in PCAP file



TLS calls vs Time

The information we can get while analysing encrypted transactions is limited, so we turned our focus to gathering decrypted data.

Performing Certificate Unpinning on the UPI Apps

Static Certificate Unpinning

First we tried a static approach to perform certificate unpinning, taking the Paytm app's apk for our experiment. Using **APK-MITM**, the APK was patched to facilitate testing with debugging tools and proxy interception. To complement this, the APK was decompiled using **Jadx**, which provided a human-readable view of the app's source code, enabling us to locate critical classes and configurations related to network security.

The investigation focused on classes such as TLSSocketFactory and VisaPubKeyManager, which are integral to establishing secure SSL/TLS connections. These classes revealed the implementation of dynamic pinning mechanisms, relying on embedded public key information to validate server authenticity. Interestingly, the app incorporated some static certificate handling, evidenced by files like signer.crt, uidai_aadhar.cer, and uidai_auth.cer. By decoding these certificates with **OpenSSL**, we confirmed their details and purpose in ensuring secure communication with backend servers.

Further analysis of classes such as CJRSSLPin and configurations related to CertificatePinner class highlighted the app's reliance on dynamic pinning mechanisms during runtime. This included checks specific to domain certificates, performed on the fly to ensure secure connections. Notably, no comprehensive client-side verification mechanisms were detected, suggesting that the server side played a dominant role in certificate validation, adding another layer of dynamic security to the app.

The app's network_security_config.xml file provided additional insights into its network security posture. This file indicated that the app disallowed cleartext traffic and relied on system-trusted certificates for its trust anchors. Such configurations aligned with modern security best practices, reinforcing the app's resilience against basic interception attempts. To complement static analysis, we captured and analyzed network traffic using **PCAPDroid** and **Wireshark**. SSL/TLS handshakes revealed the use of trusted root CAs, including "Amazon Root CA 1" and "Starfield Services Root CA." These findings confirmed the app's reliance on established trust chains during secure communication.

So now that we figured out that the app performs dynamic certificate pinning, we moved on to a dynamic unpinning approach, which required root access.

Dynamic Certificate Unpinning

A. Initial Setup and Root Detection Bypass

To begin, we rooted the Android device to enable greater control over system and app functionalities. However, rooting introduces challenges, as many payment apps implement root detection mechanisms that restrict functionality. We added the targeted apps to the **Zygisk denylist** to mask root access. This prevented immediate detection of root privileges, allowing us to access the login pages of certain apps like PhonePe and MobiKwik. However, Paytm and Google Pay continued detecting root access, rendering them unusable for our experiments.

B. Exploring Zygisk Modules for Certificate Pinning Bypass

We attempted various Magisk and Zygisk modules to bypass certificate pinning, including:

• **Always Trust User Certificates**: Forces apps to trust user-installed CA certificates.

• **MagiskHide Props Config**: Spoofs device information to avoid root detection.

• **Systemless_certificates & Systemless Hosts**: Tools to manipulate certificate trust without modifying the system directly.

• **LSPosed Framework**: Enables app-specific modifications.

Despite these efforts, the security measures in payment apps like PhonePe and MobiKwik remained intact, leading to app crashes and failed payment attempts.

C. Transition to Frida for Dynamic Unpinning

We shifted to **Frida**, a powerful dynamic instrumentation tool, to bypass certificate pinning dynamically. Using a rooted setup, Frida enabled real-time hooking into app processes without modifying APK files. Below is the detailed setup process for using Frida.

Step-by-Step Frida Setup

A. Install Frida on the Laptop:

• Ensure Python is installed and install Frida using pip install frida-tools.

• Verify the installation by running `frida --version`.

B. Set Up Frida Server on the Android Device:

• Download the compatible Frida server binary for the device architecture from the official repository.

• Transfer the binary to the device and place it in `/data/local/tmp/`.

• Set appropriate permissions: `chmod +x frida-server`

C. Start Frida Server Using Termux:

• Install Termux from the Play Store or F-Droid on your Android device.

• Open Termux and navigate to the directory where the Frida server binary is stored:

```
cd /data/local/tmp/
```

• Set executable permissions for the Frida server:

```
chmod +x frida-server
```

• Start the Frida server from Termux:

```
chmod +x frida-server
```

• Leave Termux running in the background to keep the server active.

D. Verify Connection:

• Connect the device to the laptop via USB.

• Verify the connection using `frida -U` (lists connected devices).

E. Inject the Unpinning Script:

• Obtain the dynamic unpinning script from the following link:
https://codeshare.frida.re/@akabe1/frida-multiple-unpinning/

• Obtain app the process name by using the following command with the app open on screen:
`adb shell ps | grep <app_package_name>`

• Run the injection script with the following command:

```
adb shell ps | grep <app_package_name>
```

Using Mitmproxy for Network Traffic Analysis

After successfully bypassing certificate pinning using dynamic unpinning techniques, the next step in our analysis was to capture and inspect the app's network traffic. This involved the use of **mitmproxy**, a powerful tool for intercepting and modifying HTTP/HTTPS traffic. By acting as an intermediary between the app and its backend servers, mitmproxy enabled us to observe and analyze the data transmitted during app operations.

Step-by-Step Guide to Set Up Mitmproxy on PC and Android Device

A. Setting Up Mitmproxy on PC :-

1. **Install Mitmproxy**

   • Open a terminal (or command prompt) on your PC.

   • Install mitmproxy using pip:

```
pip install mitmproxy
```

   • After installation, verify that mitmproxy is installed by running:

```
mitmproxy --version
```

   This should output the installed version of mitmproxy.

2. **Run mitmproxy** using the following command: `mitmproxy`
   By default, mitmproxy will start on port 8080. You can verify this by visiting http://localhost:8080 in your web browser.
3. **Viewing Traffic**

• Once mitmproxy is running, it will start intercepting HTTP/HTTPS traffic. Any traffic passing through the proxy will be displayed in the terminal window.

• If you want to use a specific web interface, you can use **`mitmweb.`** This opens the web interface at http://localhost:8081.

B. Setting Up Mitmproxy on the Android Device :-

1. **Install the Mitmproxy Certificate on the Android Device**

• **Download the Mitmproxy CA Certificate**:

• Open a web browser on your PC and go to the URL http://mitm.it/ . From here, you can download the appropriate certificate for your Android platform.

• Save the .crt file.

• **Transfer the Certificate to the Android Device**:

• Connect your Android device to your PC via USB.

• Transfer the .crt file to your Android device using adb push or simply copy-paste the file.

• **Install the Certificate on Android**:

• Go to **Settings** > **Security** (or **Privacy** > **Security** depending on your device).

• Tap on **Install from storage** (may vary slightly based on Android version).

• Navigate to where the .crt file was saved and select it.

• Name the certificate and then tap **OK** to install it.

• Once installed, you should see "mitmproxy" listed under trusted certificates.

2. **Configure Proxy Settings on the Android Device**

a) **Connect to the Same Wi-Fi Network**: Ensure both your Android device and PC are connected to the same Wi-Fi network.

b) **Set the Proxy**:

• Go to **Settings** > **Wi-Fi**.

• Long press on your connected Wi-Fi network and select **Modify Network**.

• Scroll down and check the box for **Show advanced options**.

• Set **Proxy** to **Manual**.

• In the **Proxy hostname** field, enter the IP address of your PC (find this by running **`ipconfig`** on Windows or ifconfig on Linux/Mac).

• In the **Proxy port** field, enter 8080 (the default port used by mitmproxy).

c) **Save the Settings** by clicking **Save**.

C. <u>Intercepting Traffic with Mitmproxy on Android</u> :-

1. **Run Mitmproxy:** In your PC terminal, ensure mitmproxy is running using **`mitmproxy.`** This should start intercepting traffic on the specified port 8080.
2. **Capture Traffic:**
   • On your Android device, open the app (e.g., Paytm, PhonePe) that you want to intercept traffic from.
   • You should now see HTTP/HTTPS requests from the app appearing in the mitmproxy interface on your PC. These requests are being routed through mitmproxy, where they will be displayed in the terminal or web interface.
   • Since you installed the mitmproxy certificate on the Android device, the SSL/TLS encryption will be bypassed, allowing mitmproxy to decrypt the encrypted traffic.

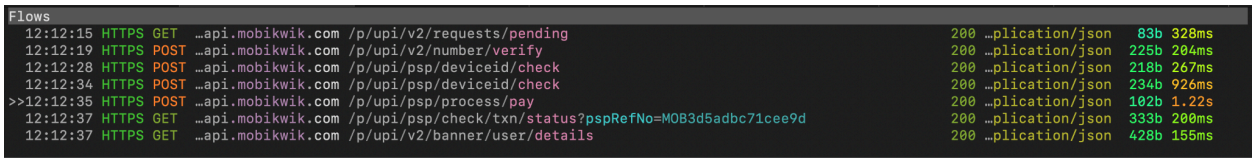
3. **Analyze Traffic:** You can now view the HTTP/HTTPS requests, inspect headers, request bodies, and response data directly within the mitmproxy interface. If you're using the web interface (mitmweb), you can also use the filters to find specific requests or responses.

A) Initial Attempt: Mitmproxy on an Unrooted Device

Initially, mitmproxy was configured on an unrooted device by installing its custom CA certificate and setting the device's proxy to route traffic through mitmproxy. This setup allowed us to intercept the app's HTTPS traffic. However, the data transmitted in the packets remained **unreadable**, as the payloads were encrypted.
The **Details** section in mitmproxy revealed only encrypted gibberish for both request and response bodies, indicating that the app was implementing additional encryption mechanisms, likely due to certificate pinning. Despite intercepting the packets, the absence of proper decryption keys rendered the content useless for analysis.

B) Role of Dynamic Certificate Unpinning

After implementing dynamic unpinning using **Frida**, the app was tricked into accepting mitmproxy's CA certificate as trusted. This allowed us to bypass the app's pinning mechanisms and capture decrypted HTTPS traffic.



Network Calls involving a UPI Transaction

Observations After Decrypting Traffic

1. Secure Communication Validation:

• The app consistently established HTTPS connections to its backend servers, verified against the pinned certificates.

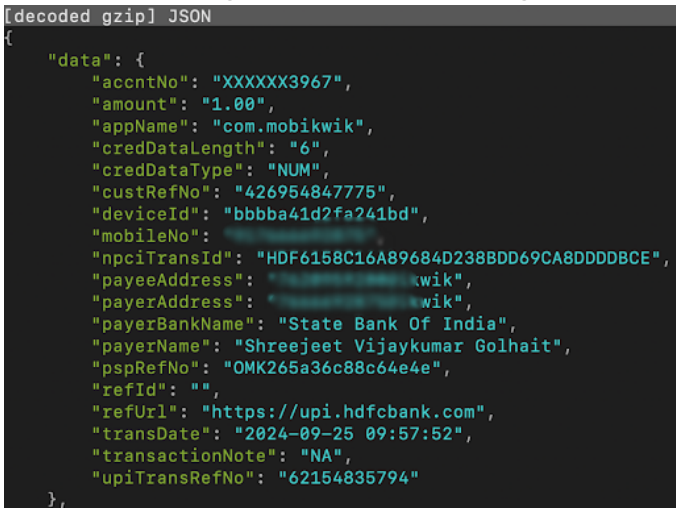
• Post-unpinning, mitmproxy revealed the full request and response bodies, providing a clear view of the transmitted data.

2. Data Exchange Patterns:

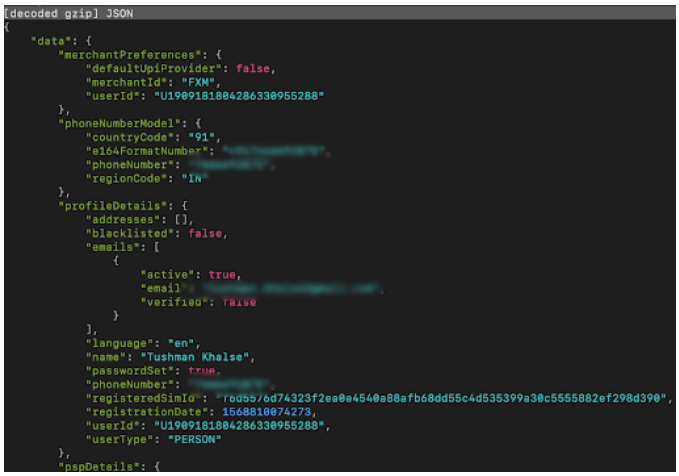
• The decrypted traffic allowed us to analyze the structure and content of the exchanged data during a UPI transaction.

• Request bodies used JSON format to encode user credentials, transaction amounts, and other inputs.

• Responses included clear acknowledgments or errors, along with user-specific data.



Payee's Information (Mobikwik)



Payer's Profile (PhonePe)

3. Use of Additional Encryption:

• While most traffic was now decrypted, certain portions of the payloads employed custom application-level encryption. These remained encoded even after bypassing pinning, reflecting an extra layer of security implemented by the app.

Implemented Automation of UPI Apps

In order to automate upi transactions, we used Appium, an open-source automation framework. It can control and interact with the mobile UPI apps by simulating user actions such as tapping, swiping, and entering text. It uses the WebDriver protocol to communicate with devices. Scrcpy was utilized for real-time mirroring and controlling the Android device on a computer. This provided a visual reference of the device's UI during automation, enhancing debugging efficiency. We have written scripts for Google Pay, Mobikwik, PayTM and PhonePe. Our scripts transfer 1 rupee on each transaction, and can be modified to do multiple transactions per execution. It also returns time taken for each transaction.

You need to have Appium, scrcpy, Node.js and other related dependencies installed for running the scripts. Android SDK is also required for automating android devices. ADB(Android Debug Bridge) is also required to connect with the android device, which is a part of the android SDK. Please ensure USB-Debugging is allowed on the mobile device.

```
npm install -g appium
```

Sometimes there can be compatibility issues between the appium-python-client and the computer's python version, hence using a python virtual environment is recommended.

```
python3.10 -m venv appium_env
source appium_env/bin/activate
pip install appium-python-client
```

The uiautomater-2 Appium driver will be necessary for the automation to work.

```
appium driver install uiautomator2
```

Now, you need to set the various dependencies required for the driver. They vary from system to system, but usually include setting the JAVA_HOME and ANDROID_HOME environment variables. Use this following command to know what dependencies must be installed for your system:

```
appium driver doctor uiautomator2
```

Appium-Inspector is used to identify which app elements are relevant. Once the Appium server is running and the android device is connected, the Appium Inspector will display the mobile screen. Hover over and select UI elements to view their properties. Appium Inspector automatically generates selectors (XPath, ID, ClassName) that you can use in your automation scripts.

The scripts need to be modified according to the android device and app it is being used on. Some parts of the script rely on screen coordinates instead of app elements. This is due to UPI apps having a secure screen flag on for some steps like entering the UPI pin. For these steps, we need to specify in the script the coordinates of the UPI pin buttons to be pressed. This will vary with the mobile device and the UPI pin to be entered, for example:

```
capabilities = dict(
      platformName='Android',
      automationName='uiautomator2',
      deviceName='OnePlus 7Pro',
      appPackage='com.mobikwik_new',
appActivity='com.mobikwik_new.ui.activity.Navigator_MainActivity',
      language='en',
      locale='US',
      noReset=True,
      fullReset=False,
      forceAppLaunch=False,
)
```

The Appium driver was configured with these capabilities, establishing a connection with the mobile device and launching the target UPI app without resetting its state. This preserved login credentials and previous session data, making the automation more efficient.

Steps automated :-

1. Contact Selection: Using Appium Inspector, UI elements (buttons, lists) were identified and interacted with. This allowed the automation script to select a contact from the app's contact list.

2. Entering Payment Details: Automated input of payment information such as contact name, amount and the UPI pin

3. Transaction Confirmation: Scripts simulated tapping the "Pay" button and confirming any subsequent confirmation prompts. The script waits for some time in order to accommodate any delays due to network conditions.

```
confirmation_message =
WebDriverWait(self.driver,15).until(EC.presence_of_element_located((A
ppiumBy.XPATH,
'(//android.widget.FrameLayout[@resource-id="com.mobikwik_new:id/head
er_container"]/android.widget.LinearLayout)[1]')))
print("Confirmation Message received")
end_time = time.time()
duration = end_time - start_time
print("Payment completed at:", time.strftime('%Y-%m-%d %H:%M:%S'))
print("Time taken between UPI PIN entry and confirmation:", duration,
"seconds")
```

4. Timestamp Capture: Recorded the start time before initiating the transaction and the end time upon receiving a confirmation. Also calculated the time difference to get the transaction duration.

Manipulating Network Conditions and Measuring Their Impact

The workflow involved network shaping, establishing a controlled network connection, transaction testing under various conditions, and collecting decrypted network data for analysis.

We used tc for network shaping. tc (Traffic Control) is a powerful Linux command-line utility used to manipulate network traffic by controlling parameters like bandwidth, latency, and packet loss. It allows for precise network condition simulation, making it ideal for performance testing under various network environments.

We had access to a router connected to the IITD network. We modified this router's connection in order to run our experiments. We had a shaper.sh script, which shaped the router's network using tc. We also wrote another script which gave the router commands regarding the exact bandwidth and latency we had to shape.

```
cmd = f"ssh root@192.168.1.1 'ash /root/shaper_dpa.sh {k}
{fixed_cap}kbit {fixed_cap+2}kbit {iface} {fixed_latency}ms
{pkt_loss_list[0]}'"
```

```
$TC qdisc add dev $IF root handle 1:0 tbf rate $RATE burst 5kb
latency 5ms peakrate $PEAKRATE mtu 2000
$TC qdisc add dev $IF parent 1:1 handle 10: netem loss $PACKET_LOSS%
```

```
delay $LATENCY $JITTER limit $LIMIT
```

Steps for Network Shaping :-

1. **Bandwidth Limitation**: Simulated low-bandwidth environments to observe app behavior under constrained conditions. The bandwidth range we used was from 100 kbps to 1000 kbps, which is what one would expect to get in India's network landscape under adverse conditions.

2. **Latency Injection**: Introduced artificial delay to simulate high-latency connections typical of poor network conditions. Our experiments ranged from 10 ms to 4000 ms.

3. **Reset network conditions**: It is recommended to also restore default settings after each test run to ensure a controlled environment for each scenario.

```
sudo tc qdisc del dev eth0 root
```

Reverse tethering enables an Android device to share a wired internet connection from a host computer, bypassing Wi-Fi. gnirehtet (reverse spelled backward) is a tool that establishes this connection over USB, making it easier to control and monitor network conditions directly from the host system. We used it to ensure the android device was connected to the internet through the router we were shaping.

One of our main goals was to evaluate how different network conditions affect transaction times and overall app performance. Here is the testing methodology:

1. **Multiple Transactions per Condition**: Conducted a series of transactions for each network configuration (normal, high latency and low bandwidth scenarios) to ensure statistical reliability.

2. **Time Measurement**: Recorded start and end times for each transaction to calculate the total transaction duration. This functionality is implemented in the automation scripts.

3. **Error Handling Observations**: Noted how the app responded to timeouts, retries, and dropped connections, providing insights into its robustness under adverse network conditions.

4. **Decrypted Data Collection**: We also logged decrypted transaction data we collected through mitmproxy. We have data corresponding to varied bandwidth and latency conditions. The data can be viewed using mitmproxy. On opening mitmproxy, press

Shift+L, then enter the path to the data log to view it. Here is the decrypted data. The logs include API request/response times, decrypted payload, error messages, payload size and more.

By simulating various network conditions using tc and reverse tethering via Gnirehtet, we thoroughly assessed the resilience and performance of UPI apps under real-world conditions. Capturing and analyzing decrypted data allowed us to understand how network fluctuations impact transaction success rates, providing actionable insights for optimizing app performance and enhancing the user experience in challenging network environments.

Analysed Collected Data and Compared Performance

To simulate real-world conditions by varying network frequencies, latencies, and bandwidth, and to observe how these factors impact transaction performance, we tested Mobikwik and PhonePe under each condition, and the results were logged. We picked these two apps as we were able to perform a successful man-in-the-middle attack against them.

Mobikwik and PhonePe were observed for transaction time variation, error-handling capabilities (timeouts, retries), success rates in high-loss environments, and we also looked out for any privacy breaches of the user's private information.

To visualize transaction performance and trends across different network scenarios and UPI apps, the collected transaction times were formatted into a structured dataset. We then plotted transaction times vs bandwidth and transaction time vs bandwidth graphs on this data. Decrypted payloads were analyzed to identify high-latency API endpoints, large payloads causing delays, and any redundant network calls contributing to inefficiency. Advertisements and trackers were also identified using the decrypted data, and if they were called during the important transaction window.

Important Insights

A. Certificate Unpinning Attempts

Static Certificate Unpinning:

• **Objective:** To bypass SSL/TLS certificate pinning by modifying methods responsible for certificate verification and rebuilding the app.

• **Outcome:** Despite installing a custom MITM (Man-in-the-Middle) certificate and modifying the certificate-checking logic, the rebuilt UPI apps failed to run properly. This indicates strong **static code integrity checks** and effective tampering detection mechanisms, preventing unauthorized modifications.

Dynamic Certificate Unpinning:

• **Objective:** To bypass certificate pinning and root detection dynamically using **Frida**, a dynamic instrumentation toolkit.

• **Outcome:** Successfully bypassed root detection in **Mobikwik** and **PhonePe**, allowing for runtime interception of network traffic. Other UPI apps demonstrated more robust anti-tampering mechanisms, making dynamic bypass attempts unsuccessful.

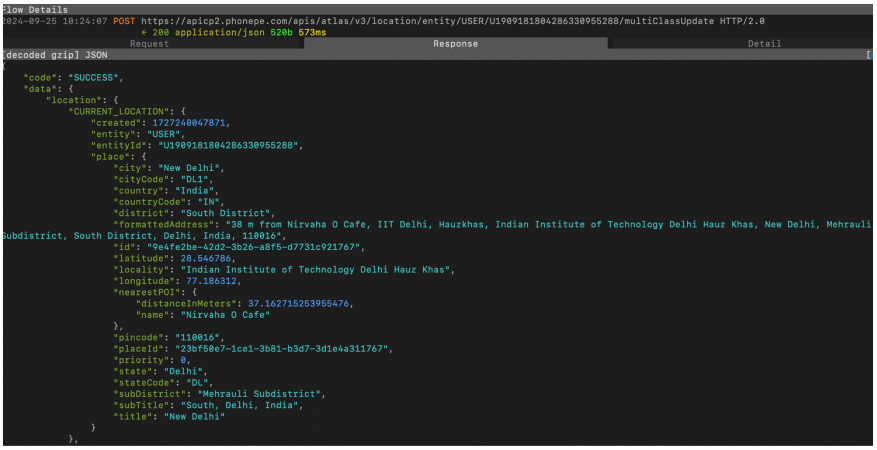
On running **mitmproxy** during UPI transactions with Mobikwik and PhonePe, we figured out the exact network calls corresponding to the network calls in a payment transaction.

B. App-Specific Network Behavior

PhonePe:

• **Excessive Advertisement Calls:** During payment transactions, PhonePe generates a high volume of **advertisement-related network calls**. These calls consume network resources, potentially **slowing down the transaction process**.

• **Privacy Concern:** The app collects **user location data** primarily for ad purposes, even during payment transactions, raising concerns about unnecessary data collection.



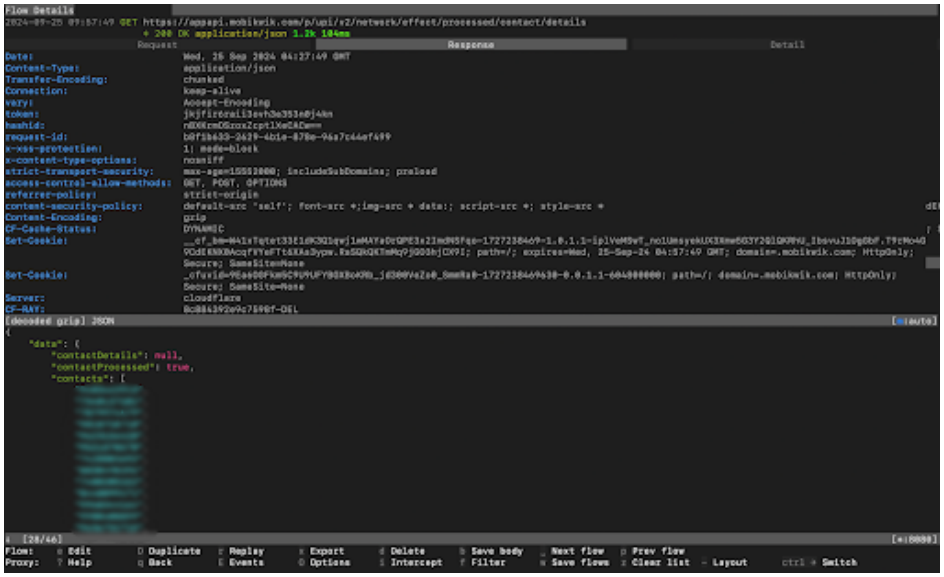
Payer's Accurate Location being sent to PhonePe server

Mobikwik:

• **Contact List Transmission:** Mobikwik sends the **entire user contact list** to its server during every transaction. This introduces:

• **Privacy Risks:** Sharing sensitive user information without explicit consent.

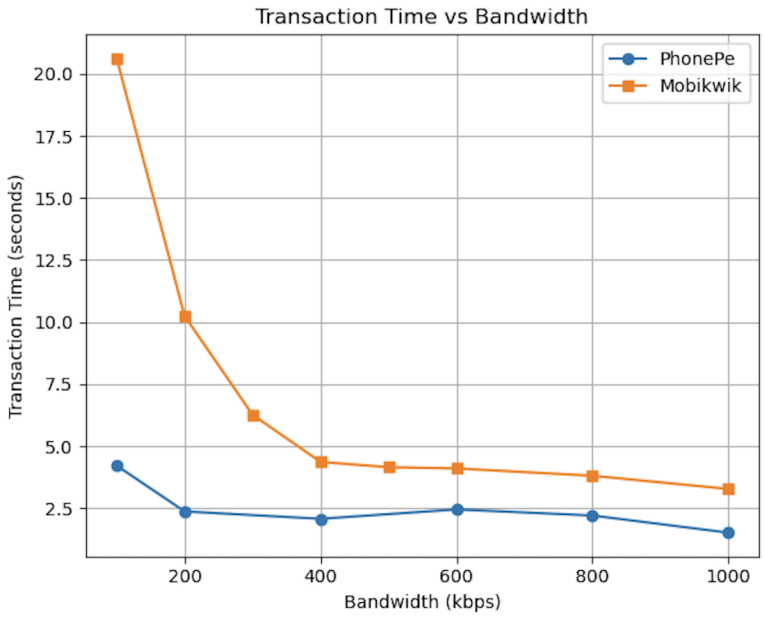
• **Network Load:** Increased data transmission size, leading to higher bandwidth usage, especially in limited bandwidth scenarios.

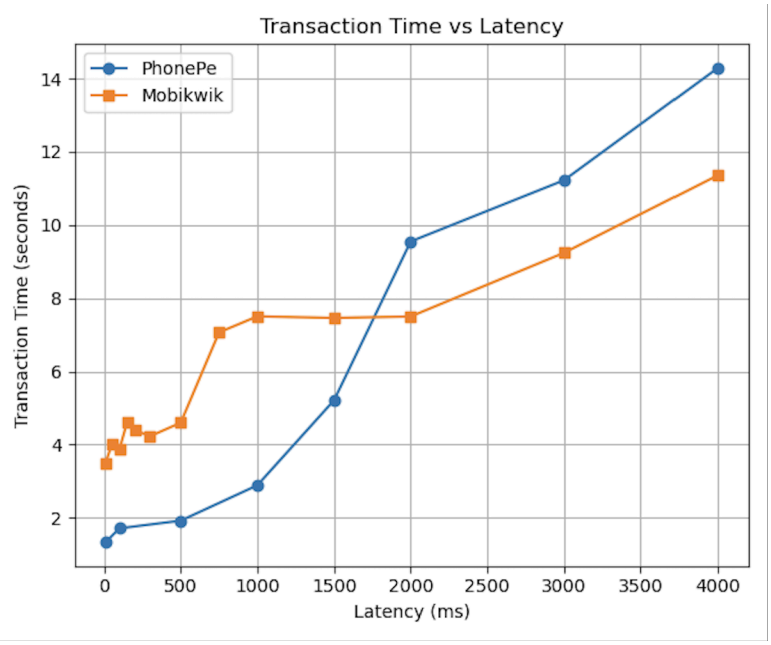


Payer's Entire Contact List being sent to Mobikwik Server

C. Network Scenarios Simulated

• **Transaction Times:** Increased significantly under poor network conditions, with a noticeable rise in **transaction failure rates** in extreme scenarios (e.g., packet loss > 20% or latency > 1000ms).





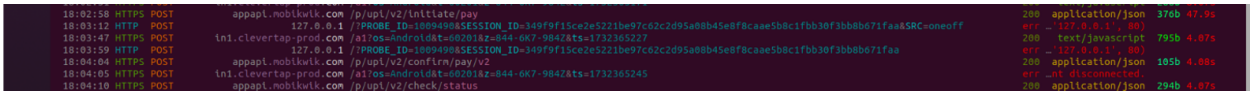
• **App-Specific Findings:**

Mobikwik: Showed **poor performance** in limited bandwidth scenarios, suggesting **inefficient prioritization** of critical payment-related network calls.

PhonePe: Performed **better under limited bandwidth**, maintaining a **lower average transaction time** compared to Mobikwik, thanks to more efficient handling of network resources and a more efficient network protocol.

D. Traffic Prioritization Analysis

Advertisement and Tracker Calls:



Advertisement network calls in the middle of a UPI transaction at very high latency (PhonePe)

Both apps continued to make **advertisement and tracker calls** during payment transactions, even under adverse network conditions. However:

PhonePe: Demonstrated **better prioritization** of essential payment-related calls over ad-related calls, leading to improved performance.

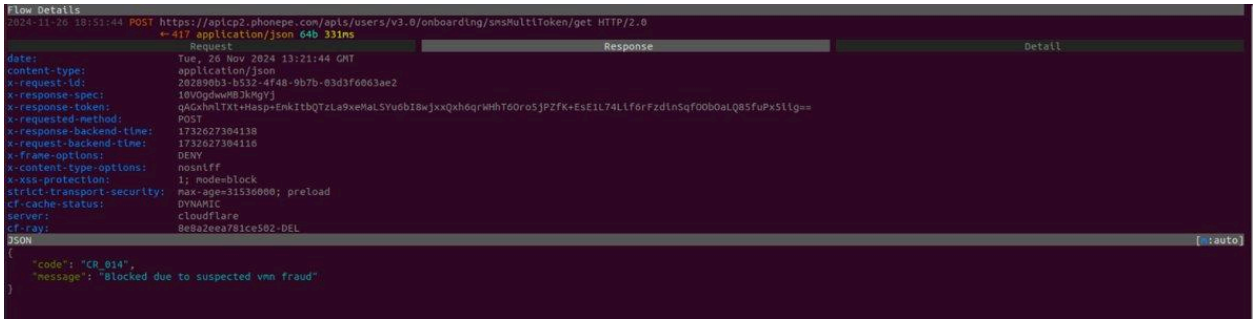
Mobikwik: Did not deprioritize non-essential calls effectively, contributing to slower transactions in limited bandwidth environments.

Details of an advertisement network call in PhonePe

Specific API calls (e.g., OTP validation, transaction confirmation) were correlated with slow transaction times. For example, repeated requests to a payment gateway API under poor network conditions were identified as a bottleneck.

E. Malpractice Detection and Response

PhonePe : Detected suspicious activity related to our automation and analysis efforts, leading to a **temporary account ban**. This indicates robust mechanisms for detecting potential malpractice or automated interactions.



Network Call corresponding to alert "Unable to Proceed due to Security Reasons" in PhonePe

Mobikwik : Mobikwik failed to identify any abnormal activity during the same tests, revealing **weaker security measures** for detecting automated interactions or potential misuse.

F. Challenges Faced

Transaction Limits: UPI Apps usually have a 20 transactions per day limit, which inhibits our capacity to take more readings.

Root Detection by Popular Apps: Security measures in popular apps like GPay and Paytm detected root access, preventing further analysis.

MITMProxy Traffic Capture: Mitmproxy captures only encrypted traffic on non-rooted devices, restricting the ability to capture unencrypted traffic without root access.

Flagging by PhonePe: PhonePe detected malpractice amidst experiments, banning the device and halting further testing.

Wireless Dependency for Mitmproxy: Mitmproxy requires a wireless connection setup and cannot be used effectively in wired configurations, adding constraints to the testing environment.

Future Scope

A. Frida Unpinning and MITMProxy

The integration of Frida for dynamic certificate unpinning and mitmproxy for network traffic interception presents an opportunity to explore the network calls of any highly secure application, not just limited to digital payment apps. This combination can be used on rooted Android devices to perform detailed analysis of secure communication in a broader range of apps, such as banking applications, social media platforms, or any app utilizing sensitive data exchange. By dynamically unpinning certificates, the unencrypted data can be captured and analyzed, providing a deeper understanding of the app's communication patterns and potential vulnerabilities.

B. Automation of UPI Payments without root access

The current experiment required root access for dynamic certificate unpinning. However, the future scope could involve developing automation scripts that bypass the need for root privileges, enabling the automation of security testing processes. These scripts could facilitate experiments involving multiple consecutive UPI payments on popular apps like Paytm, Google Pay, PhonePe, and MobiKwik. By introducing minor modifications, the scripts could be adapted to automatically intercept and analyze network traffic related to payment transactions, reducing manual intervention and allowing for large-scale testing of these apps' security features over time.

C. Investigation into UPI Apps' Backend Security Mechanisms

While the project focused on front-end security aspects such as certificate pinning, an area for future research could be the exploration of UPI applications' backend security mechanisms. This includes investigating the security protocols used in server-client communication, encryption methods, and how UPI apps handle sensitive data such as authentication tokens, payment details, and personal information. Additionally, exploring how UPI apps handle third-party integrations (e.g., bank APIs, QR code scanners) could reveal potential vulnerabilities in the interaction between different components of the payment system.

D. Further Network Analysis

Another valuable direction for future research would be to conduct more extensive testing of UPI apps under varied network conditions. By modifying network parameters such as packet

loss, and jitter, we can simulate real-world scenarios where network quality is unstable or suboptimal. Additionally, testing the apps' behavior when switching between different types of networks, such as 4G, Wi-Fi, and even slow or unreliable mobile data connections, would provide insights into their robustness and resilience in less-than-ideal circumstances, leading to the development of more fault-tolerant systems that can ensure smooth payment experiences even under challenging network conditions.

E. Cross-Platform Analysis

Expanding the research to analyze UPI applications across different platforms—Android, iOS, and web—would provide a comprehensive understanding of security practices in multi-platform environments. Differences in how secure communications are handled on different operating systems could reveal platform-specific vulnerabilities or advantages, aiding developers in building more robust cross-platform UPI applications.

F. Regulatory and Compliance Aspects

Another critical area for future research involves studying the regulatory and compliance aspects of UPI security. Understanding how UPI apps align with government regulations, such as those set by the Reserve Bank of India (RBI), could offer insights into the legal and ethical challenges of testing the security of digital payment systems. This would ensure that security testing aligns with the necessary compliance standards, protecting users' privacy and safety while adhering to legal frameworks.

By building on the findings and methods established in this project, future research can enhance our understanding of UPI app security and contribute to the development of more secure, efficient, and resilient mobile payment systems.

References:

1. Initial guidance to analyse UPI apps with some useful guidelines: https://faculty.iiitd.ac.in/~arani/assets/pdf/eurosp24.pdf
2. Script for dynamic certificate unpinning: https://codeshare.frida.re/@akabe1/frida-multiple-unpinning/
3. Script for network shaping using tc: https://gist.github.com/jterrace/1895081