Security Analysis of Unified Payments Interface and Payment Apps in India

Renuka Kumar, *University of Michigan;* Sreesh Kishore*;* Hao Lu and Atul Prakash, *University of Michigan*

https://www.usenix.org/conference/usenixsecurity20/presentation/kumar

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

Security Analysis of Unified Payments Interface and Payment Apps in India

Renuka Kumar[1], Sreesh Kishore , Hao Lu[1], and Atul Prakash[1]

[1]*University of Michigan*

Abstract

Since 2016, with a strong push from the Government of India, smartphone-based payment apps have become mainstream, with over $50 billion transacted through these apps in 2018. Many of these apps use a common infrastructure introduced by the Indian government, called the Unified Payments Interface (UPI), but there has been no security analysis of this critical piece of infrastructure that supports money transfers. This paper uses a principled methodology to do a detailed security analysis of the UPI protocol by reverse-engineering the design of this protocol through seven popular UPI apps. We discover previously-unreported *multi-factor authentication* design-level flaws in the UPI 1.0 specification that can lead to significant attacks when combined with an installed attacker-controlled application. In an extreme version of the attack, the flaws could allow a victim's bank account to be linked and emptied, even if a victim had never used a UPI app. The potential attacks were scalable and could be done remotely. We discuss our methodology and detail how we overcame challenges in reverse-engineering this unpublished application layer protocol, including that all UPI apps undergo a rigorous security review in India and are designed to resist analysis. The work resulted in several CVEs, and a key attack vector that we reported was later addressed in UPI 2.0.

1 Introduction

Payment apps have become a mainstream payment instrument in India, with the Indian Government actively encouraging its citizens to use electronic payment methods after a demonetization of large currency notes in 2016 [29]. To facilitate digital micro-payments at scale, the National Payments Corporation of India (NPCI), a consortium of Indian banks, introduced the Unified Payment Interface (UPI) to enable free and instant money transfers between bank accounts of different users. As of July 2019, the value of UPI transactions has reached about $21 billion [45]. UPI's open backend architecture that enables easy integration and interoperability of new payment apps is a significant enabler. Currently, there are about 88 UPI payment apps and over 140 banks that enable transactions with those apps via UPI [40, 41]. This paper focuses on vulnerabilities in the design of UPI and UPI's usage by payment apps.

We note that hackers are highly motivated when it comes to money, so uncovering any design vulnerabilities in payment systems and addressing them is crucial. For instance, a recent survey states a 37% increase in financial fraud and identity theft in 2019 in India [12]. Social engineering attacks to extract sensitive information such as one-time passcodes and bank account numbers are common [17, 23, 34, 57, 58].

Payment apps, including Indian payment apps, have been analyzed before, with vulnerabilities discovered [9, 48], and an Indian mobile banking service was found to have PIN recovery flaws [47]. However, in these studies, mobile apps did not share a common payment interface. As far as we are aware, an analysis of a common interface used by multiple payment apps has not been done before. Such an analysis is important because security flaws in them can impact customers of multiple banks and multiple apps, regardless of other stronger security features used. We focus on the security analysis of the unified payment interface used by many Indian payment apps and its design choices.

In this work, we use a principled approach to analyze UPI 1.0, overcoming significant challenges. A key challenge is that the protocol details are not available, though millions of users in India use it. We also did not have access to the UPI servers. We thus had to reverse-engineer the UPI protocol through the UPI apps that used it and had to bypass various security defenses of each app, including code obfuscation and anti-emulation techniques. Though we build on techniques used in the past for security analysis of apps [9, 21, 46, 48], our approach to extract the protocol details varies based on the defenses the apps use. We carefully examine each stage of the UPI protocol to uncover the credentials required to progress in each stage, find alternate workflows for authentication, and discover leakage of user-specific attributes that could be useful at a later stage.

We present results from the analysis of the UPI protocol,

App Name	Launched	Versions	Installs	Rating	UPI
BHIM	Dec, 2016	1.3, 1.4, 1.5	10M+	4.1	1.0
Ola Money	Nov, 2015	1.8.1, 1.8.2, 1.9.0	1M+	3.8	1.0
Phonepe	Dec, 2015	3.0.6, 3.3.23	100M+	4.5	1.0
SamsungPay	Aug, 2016	2.8.49, 2.9.3	50M+	4.7	1.0
Paytm	Aug, 2010	8.2.12	100M+	4.4	2.0
Google Pay (Tez)	Sept, 2017	39.0.001	100M+	4.4	2.0
Amazon Pay[1]	Feb 2019	18.15.2			2.0

[1]Amazon Pay is not available on Google Play store

Table 1: List of apps analyzed and their Google Play ratings

as seen by seven of the most popular UPI apps in India listed in Table 1. Of the seven apps we analyze, four UPI apps—Google Pay (Tez), PhonePe, Paytm, and BHIM—have a combined market share of 88% [27] and are widely accepted at many shopping sites. From a total of 88 UPI apps, many are minor variations of BHIM, the flagship app released by NPCI (also the designers of UPI). Close to 48 banks today issue a bank-branded version of the BHIM app. Since Android owns over 90% of the Indian mobile market share [13], we focused on the Android versions of these apps.

Our threat model assumes that the user is careful to use an authorized payment app on a non-rooted Android phone, but has installed an attacker-controlled app with commonly used permissions. We also do not rely on the success of social engineering attacks, though they could simplify exploiting some of the vulnerabilities we uncovered. We uncovered several design choices in the UPI 1.0 protocol that lead to the possibility of the following types of attacks:

- *Attack #1: Unauthorized registration, given a user's cell number:* This attack leaks private data such as the set of banks where a user has bank accounts and the bank account numbers.
- *Attack #2: Unauthorized transactions on bank accounts given a user's cell number and partial debit card number:* Purchases using a debit card in India, whether in-store or online, requires a user to authorize the payment by entering a secret PIN. In this attack, an attacker, by knowing a user's cell number and debit card information printed on the card (last six digits and expiry date, without the PIN), can do transactions on a bank account of a user who has never used a UPI app for payments.
- *Attack #3: Unauthorized transactions without debit card numbers:* This attack shows how an attacker that starts out with no knowledge of a user's authentication factors can learn all the factors to do unauthorized transactions on that user's bank account.

Our work started over two years ago when NPCI released UPI 1.0 and BHIM, which are the focus of our analysis. Given the potential risks with releasing our findings, we waited to publish until NPCI addressed a critical attack vector in the recently released UPI 2.0. Our key contributions are as follows.

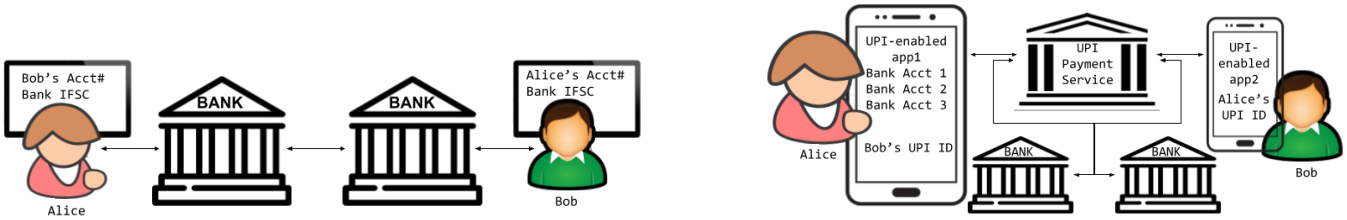
- We conduct the first in-depth security analysis of the unpublished UPI 1.0 protocol that provides a common payment interface to many popular mobile payment apps in India and allows bank-to-bank transfers between users of different apps.
- We show how to systematically reverse-engineer this complex application layer protocol from the point-of-view of an adversary with no access to UPI servers. We use BHIM, the reference implementation for UPI apps released by the Indian government, for our initial analysis and then confirm our findings on other UPI apps.
- We found subtle design flaws in the UPI protocol, which can be exploited by an adversary using an attacker-controlled app that leverages known flaws in Android's design, to construct scalable remote attacks. We show how an adversary can carry out the attacks starting with no knowledge of a user.
- As responsible disclosure, we reported the flaws to app developers, CERT India, and CERT US, resulting in several CVEs. A key attack vector we reported to NPCI and CERT India was addressed in UPI 2.0.
- We present early findings from an ongoing analysis of UPI 2.0, using BHIM, Google Pay, Amazon Pay, and PayTM—four top-rated UPI 2.0 apps in India. Findings indicate that some vulnerabilities remain.
- We discuss lessons learned and potential mitigation strategies to consider when designing such protocols.

2 Background

Early mobile payment apps in India were wallet-only apps. They could withdraw money from a user's bank account by asking a user to enter a debit card number, but not deposit money back into the bank account. Post demonetization (in 2016), to encourage cashless transactions, a consortium of Indian banks called the National Payments Corporation of India (NPCI), backed by the Indian government, introduced the Unified Payments Interface (UPI) that allows NPCI-certified mobile apps to do free instant money transfers between bank accounts of different users. UPI apps can inter-operate with each other since they all share the same payment interface. A user of BHIM, for instance, can transfer money instantly for a small purchase from her bank account to the bank account of a shopkeeper who uses Google Pay. Because of this, most stores in India accept mobile payments through UPI apps. Depending on the app, a user can do unlimited transactions up to $1500 per transaction. Figure 1b shows the UPI money transfer system when compared with the traditional Internet banking system in Figure 1a.

2.1 User Registration on a UPI App

The UPI payment system requires Alice to register her primary cellphone (or cell) number with her bank account(s)



(a) Internet

(b) UPI-based

Figure 1: Internet vs. UPI-based Money Transfer

out-of-band to send or receive money. UPI uses the cell number (i) as a proxy for a user's digital identity with the bank to look up a bank account given a cell number; (ii) as a factor in authentication via SMS one-time passcodes (OTP); and (iii) to alert users on transactions. The Government of India requires cellphone providers to get copies of government-issued IDs, manually verify the IDs, and do biometric verification before issuing a cell number [1].

To register for UPI services, Alice must set up her UPI user profile, add a bank account, and enable transactions on that bank account, as follows:

1. *Set up a UPI user profile:* Alice must first create a profile with UPI via a UPI app installed on her bank-registered cell phone. Alice must first give her cell number to UPI through the UPI app for verification. How UPI collects this information from a user may change with each app. For instance, some apps read the cell number from the device, while others ask the user to key-in the cell number. For instance, Figure 2, screenshot #3, shows how BHIM reads Alice's cell number(s) from her phone for Alice to choose from. The UPI app then sends Alice's cell number to the UPI server for verification. Once verified, the UPI server issues a UPI ID for Alice on that app. Figure 2, screenshot #4 shows how BHIM notifies Alice when she is verified. If Alice uses multiple apps, the UPI server issues a different UPI ID for each app. The app then prompts Alice to set a passcode. The nature of the passcode is again specific to the app. BHIM, for instance, asks the user to set a 4-digit passcode, as shown in Figure 2, screenshot #5.

2. *Add a bank account:* Once Alice's profile is set up, she must add the bank account that she wants to use for withdrawals and deposits. Alice is given a list of bank names that support UPI (Figure 2, screenshot #6), from which she can now choose her bank. Alice may repeat this step to add multiple bank accounts.

3. *Enable transactions:* For Alice to be able to transact on an added bank account, she has to set up a UPI PIN for that account before the first transaction. The UPI PIN is Alice's secret to authorize any future transactions. To set the UPI PIN, Alice must furnish information printed on the debit card— the last six digits of her bank's ATM or debit card number and expiration date. Alice must also enter an OTP she receives from the UPI server. The UPI PIN is a highly sensitive factor since the UPI server uses it to prevent unauthorized transactions on Alice's bank account.

To transfer money to Bob, Alice first logs into a UPI app using the passcode she set during user registration. Then, out-of-band, Alice requests Bob to provide his UPI ID, which is often Bob's cell number. Alice chooses one of the bank accounts she previously added to the app (Figure 2, screenshot #7), initiates the transaction to Bob, and authorizes it by providing her UPI PIN. Internally, the UPI payment interface directly transfers money from Alice's chosen bank account to Bob's bank account linked with his UPI ID.

2.2 UPI Specs for User Registration

The UPI specifications released by NPCI [44] provide "broad guidelines" on the client-server handshake between a UPI app and the UPI server. We discuss the protocol details available to us from the specification.

1. *Set up a UPI user profile:* Once a UPI app gets a user's cell number, the app must send an outbound encrypted SMS from Alice's phone to the UPI server. This process is automated and does not involve the user in order to guarantee a strong association between a user's cell phone and her device. According to UPI, this is the "most critical security requirement" of the protocol since all money transactions from a user's device are first verified based on this association. UPI calls this association of a user's device (identified by parameters such as Device ID, App ID, and IMEI number) with her cell number as *device hard-binding*. The combined cell number and device information (that represents this binding) is called the *device fingerprint*, which per the UPI spec is the first factor of authentication.

[1] A recent Indian Supreme Court ruling forbids Aadhar's biometric verification for issuing cell numbers. The impact of that ruling on UPI-based apps and banks is yet to be seen, as it may make it easier for an attacker to do an unauthorized transfer of a cell number and then take over an account. We do not discuss this attack vector in this paper.

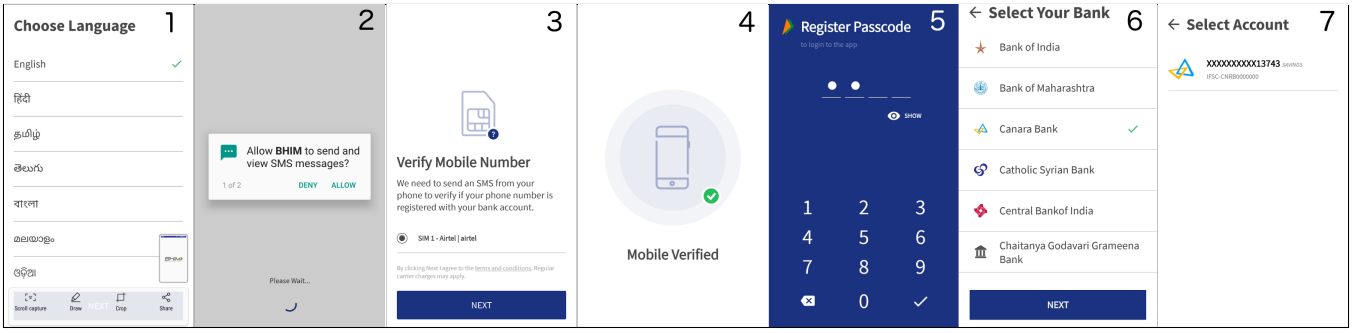


Figure 2: BHIM User Registration Using 3FA

Passcode. The UPI spec considers application passcode as optional and does not undertake responsibility for passcode authentication. UPI leaves it up to a UPI app vendor to authenticate the passcode. Thus, the responsibility to completely authenticate a user is shared between two servers— the UPI server (that verifies device fingerprint and UPI PIN), and a payment app server (that verifies an app passcode).

2. *Add a bank account:* A user's request to add a bank must be from the device registered with UPI. Internally, UPI fetches the chosen bank's account number and IFSC code based on a user's cell number for later transactions through the UPI app.

3. *Enable transactions:* UPI allows transactions to be done either using a cell number or an account number and IFSC code or any UPI ID. UPI spec mandates that all transactions must at least be 2FA using a cell phone (the device fingerprint) as one factor and the UPI PIN as the second. The spec considers a cell phone as a "what you have" factor, which allows UPI to provide "1-click 2-Factor Authentication" using the said two factors.

For apps that integrate with UPI, NPCI enforces application security via a code review and certification process. All communication with the UPI server is over a PKI-based encrypted connection. Currently, UPI has become the de facto standard for mobile transactions.

2.3 Threat Model

We assume a normal user, Alice, who installs payment apps from official sources such as Google Play; none of the payment apps contain extraneous malicious code. Alice has a properly configured phone with Internet facility and prevents physical access to it by untrusted parties.

On the other hand, the attacker, Eve, uses a rooted phone. Eve can use any tool at her disposal to reverse engineer the payment apps. We assume that Eve releases an apparently useful unprivileged app called *Mally* that requests the following two permissions—android.permission.INTERNET and android.permission.RECEIVE_SMS. Alice finds the app useful and installs it, granting it the necessary permissions.

The permissions requested for Mally are not unusual for Android. Recent versions of Android automatically grant the INTERNET permission without a user prompt [15]. SMS permissions have legitimate uses on Android, and about 15% of the Android apps request them [20]. RECEIVE_SMS permission only grants the permission to read incoming SMS messages, but not read previously received messages or send SMS messages. This permission is used by many popular social media apps such as Telegram and WhatsApp, SMS/-call blocker apps, and also security apps such as Kaspersky Mobile Security and BitDefender.

We consider our threat model to be realistic for the following reasons. First, according to the Android security review for the last two years, India is among the top three countries with the highest rate of potentially harmful applications such as trojans and backdoors, sometimes pre-installed on Android devices [24, 25]. Google has also recently released a warning stating that 53% of the major attacks are because of malicious apps that come pre-installed on low-cost smartphones [19].

To simplify some attack descriptions, we describe Mally with the READ_PHONE_STATE or accessibility permissions. We do this to show the many ways an adversary can get a user's information, e.g., a user's cell number. However, in such cases, we also show other attack vectors that require neither of these two permissions.

3 Security Analysis

3.1 Methodology

In this section, we describe how we reverse-engineer UPI, a proprietary protocol, to learn its authentication handshake. Since we do not have access to UPI's servers, we choose to reverse engineer this application layer protocol through the payment apps that support it.

Protocol Analysis. To reverse-engineer UPI, we first uncover each step of the client-server authentication handshake

with the goal of (i) understanding how UPI does device fingerprinting; and (ii) establishing the credentials required by a user to set up an account and do transactions. Besides UPI's default authentication workflow, we also look for alternate workflows or paths that could be leveraged to minimize the credentials required by an attacker. Finally, we look for any leaked user-specific attributes during protocol interactions that could be leveraged later, if intercepted, by an adversary. We triage our findings from different workflows to find plausible attack vectors and to verify potential exploits.

The approach we use to extract protocol data varies based on the specifications of an app and the security defenses they use. Since UPI 1.0 specs only state broad security guidelines rather than protocol details, we examine multiple apps to know whether the protocol varies across different apps. We analyze BHIM, the flagship app published by the same government organization that maintains the UPI system and then confirm our findings by analyzing additional apps.

App Reversing-Engineering. One approach to capture the protocol data sent and received by an app is to run it in a sandbox. Sandbox tools such as CuckooDroid [14] use an emulator for dynamic analysis. Hence, to test if the UPI apps can run in a sandbox, we manually run each app in Android SDK's built-in emulator on a Linux host. However, we find that these apps do not run without a physical SIM card, which is unavailable on an emulator. The apps also use anti-emulation techniques that prevent them from running in an emulator.

Besides anti-emulation, we find that the payment apps also use several other defenses. For instance, all of them detect a rooted phone and deter a user from running the app on a rooted phone. Some apps also look for the presence of hooking libraries such as *Xposed* [28] that typically require root access to modify system files. That apart, all apps are obfuscated, use encrypted communication, enforce session timeout and account lockout, avoid storing or transmitting data in the clear, and avoid using hard-coded credentials or keys. The extent of security defenses used by these apps shows that app developers have designed the apps with security in mind. This is unlike findings by Reaves et al. [48] that found basic security flaws in Indian payment apps around 2015.

Our security assessments show that some apps, such as BHIM, allow repackaging. We leverage this to instrument an app's code statically to learn specifics of the authentication handshake, such as the name of the activity and method that generated network traffic. Because such specifics help with precise analysis, we first check whether the apps can be instrumented and repackaged. To instrument the app, we first disassemble it using *APKTool* [4], insert debug statements, and then repackage it with our signature.

One question that arises is where to instrument in an app's code as this requires knowledge of the methods of the app we want to instrument. Since we do not know this *a priori*, we manually reverse-engineer the apps using the *JEB* [30] disassembler and decompiler. Some times, JEB fails to decompile certain classes that are control-flow obfuscated. In such cases, we use JDK's *javap* command to read bytecode. We augment our analysis with results from the static components of two hybrid analyzers *MobSF* [21] and *Drozer* [26].

We could not repackage certain apps such as Google Pay. In such cases, we intercept an app's network traffic using a TLS man-in-the-middle proxy called *mitmproxy* [36]. We install the OpenVPN app on our Android phone and an OpenVPN service on a Linux host and configure the host's firewall rules to route traffic to the *mitmproxy*. The setup also requires that we install *mitmproxy's* certificate on the phone. However, we find that starting Android Nougat, Android does not trust user-installed certificates, and setting up a system certificate requires root access, an impediment. Hence we conduct our analysis on Android Marshmallow and Lollipop devices.

3.2 Analysis of BHIM & UPI 1.0 Protocol

Bharat Interface for Money (BHIM) [5] is the Indian government's reference implementation of a payment app over UPI and was launched along with UPI 1.0. We discuss findings from our analysis of BHIM's user registration process for a user Alice whose UPI ID is her cell number. We instrument BHIM to see the protocol data it exchanges with the UPI server during registration. We show an example of how we instrument BHIM in the Appendix.

3.3 BHIM User Registration Protocol

Steps 1-10 on the left of Figure 3 are the steps of the client-server handshake between BHIM version *1.3* and the UPI 1.0 server, with minimal and relevant protocol data shown. The screen numbers (circled) on the left indicate the screenshot of the app in Figure 2 that generated the traffic. We describe the ten steps of UPI's *default workflow* below.

1. Step 1: When Alice starts BHIM, BHIM first requests Alice permission to send SMS messages (for later use) (Figure 2, #2). Once BHIM gets the permission, BHIM sends Alice's device details such as the device's Android version, device ID, make, manufacturer, and model to the UPI server as an HTTPS message.

2. Step 2: UPI server sends Alice a 13-digit registration token that identifies her device and waits to get the token back from Alice as an SMS message.

3. Step 3: BHIM app sends the registration token as an SMS message to the UPI server. BHIM waits for SMS delivery confirmation using the *sendTextMessage* API's *deliveryIntent*.

4. Step 4: When the UPI server receives the SMS, it (i) learns that Alice got the token; and (ii) gets her cell number from the message. The UPI server uses this information to hard-bind Alice's cell number to her device.

UPI server also sends a confirmation to BHIM that it received the SMS.

5. Step 5: BHIM requests a status of its device's hard-binding from the UPI server by sending the registration token back to the server as an HTTPS message.

6. Step 6: The UPI server responds with a verification status that includes Alice's customer ID, a registration token, etc. back to Alice. By now, the UPI server has verified both Alice and her device (Figure 2, #4).

7. Step 7: BHIM asks Alice to set a passcode (Figure 2, #5). The app concatenates the SHA-256 hash of Alice's passcode with her cell number and sends it as an HTTPS POST request to the UPI server.

8. Step 8: The UPI server issues a login token to Alice (BHIM), which confirms that her profile is setup.

9. Step 9: BHIM then shows Alice a list of banks that support UPI (Figure 2, #6). When Alice chooses her bank from this list, BHIM sends a bank ID to the UPI server.

10. Step 10: The UPI server sends Alice's bank account details such as her masked account number, the hash of the account number, bank name, IFSC code, etc. back to BHIM (Figure 2, #7).

The protocol description until now has seen two factors—a) cell phone (and hence a device fingerprint) as required by the UPI spec; b) a secret passcode— both of which BHIM sends to the UPI server during the handshake. For BHIM, this means that the payment app server that authenticates a user's passcode and the UPI server that verifies a device's fingerprint is the same, a fact that is not surprising since the designers of UPI also wrote BHIM.

Finally, to enable transactions, Alice sets a UPI PIN on her bank account for which she needs her bank's debit card number and expiry date, as mentioned in Section 2.1.

Alternate Workflow1. In the default workflow described above, BHIM sends the device registration token to the UPI server as an SMS message for device hard-binding (Step 3). In case the UPI server does not receive the SMS, thus failing to hard-bind, BHIM provides an alternate workflow for hard-binding, as shown in Figure 4a. BHIM prompts Alice to key-in her cell number; BHIM sends the keyed-in cell number along with the device registration token to the UPI server as an HTTPS message. The UPI server sends an OTP to Alice, which she must enter to complete device binding. The remainder of the protocol proceeds as before.

Alternate Workflow2. If Alice, an already registered user, changes her cell phone, then the UPI server has to re-bind her cell number with the new cell phone. At the time of device binding, the UPI server finds that an account for Alice already

Figure 3: BHIM User Registration Default+

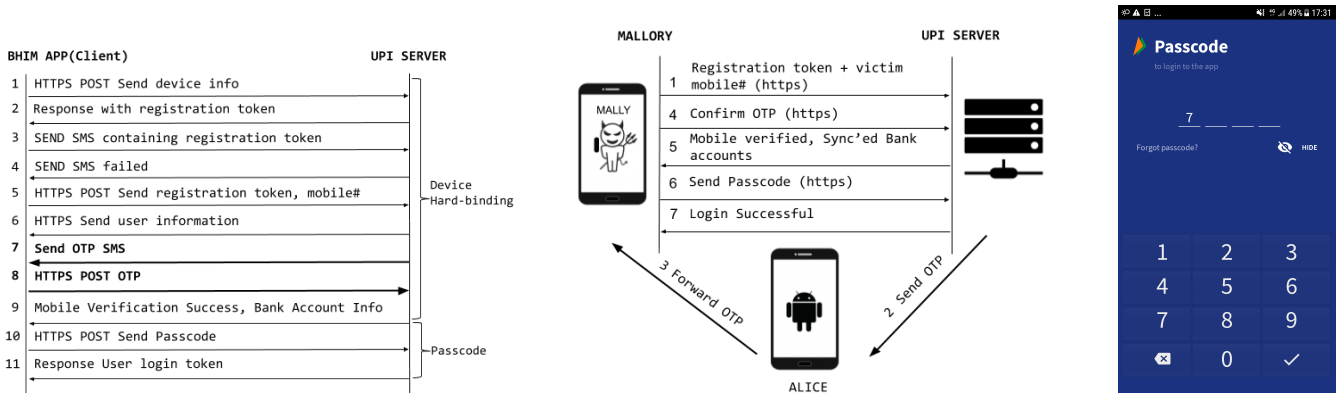
+BHIM masks bank account number in step 10 of the handshake. The authors masked the other info to safe-guard privacy.

exists and notifies BHIM of the same (*accountExists* flag in Step 6). The UPI server prompts Alice for her passcode, and once Alice is verified (Step 7), the server sends back Alice's bank account information that she previously added to BHIM (Step 10). This workflow makes it convenient for Alice to transfer her bank accounts to another phone, without going through the hassle of adding all her bank accounts again.

3.3.1 Potential security holes—initial analysis

Before we describe the attacks on the UPI protocol, we first discuss three potential security holes that we observe:

1. *Potential Security Hole #1:* For an attacker Eve to take over Alice's account, one of the first barriers to overcome



(a) BHIM Alternate Handshake (b) BHIM Attack Overview (c) Overlay Attack

Figure 4: BHIM Alternate Handshake & Attack

is UPI's device binding mechanism that binds Alice's cell number with her cell phone. For Eve to break the binding, Eve must able to bind her cell phone with Alice's cell number. Though the default workflow makes this hard, the *alternate workflow1* provides a potential fallback that allows Eve to send Alice's cell number as an HTTPS message from Eve's phone.

2. *Potential Security Hole #2:* The *alternate workflow1* uses OTP verification for device-binding. If Alice, say, enters a friend Bob's cell number on her phone, the UPI server will send the OTP to Bob's phone. If Bob shares that OTP with Alice, then Alice can confirm the OTP to the UPI server, which will hard-bind Alice's phone to Bob's cell number. As a result, Bob will receive all future SMS messages sent by the UPI server to Alice.

3. *Potential Security Hole #3:* In UPI's default workflow, Alice at no point provides a secret that she shares with her bank to confirm her identity. Nevertheless, the UPI server reveals an existing user Alice's account details in the *alternate workflow2*.

None of the security holes by themselves are exploits as yet. Below we discuss the potential attacks as a result of these holes.

3.3.2 Attack #1: Unauthorized registration, given a victim's cell number

In this attack, we show how a remote attacker, Eve, can set up a UPI account, given a victim's cell number. For the attack to succeed, Eve requires only one thing: the victim's cell phone to have *Mally* app installed.

The attack setup is as follows. Eve on her phone has a repackaged version of BHIM that has client-side security checks disabled. Eve sets up a command and control (C&C) server, puts out Mally as a potentially useful app on various app stores, and waits for unsuspecting users to install Mally. As discussed in the Threat Model (Section 2.3), Mally has RECEIVE_SMS permission. An unsuspecting user Alice, uses a legitimate version of BHIM on a non-rooted phone, as is the best practice for Android.

For attacks to happen, Eve must have a way to discover a victim's cell number. To simplify the attack description, we assume that Mally also has READ_PHONE_STATE permission, which it uses to get the cell number from the victim's phone (almost 35% of the apps use this permission [60]). We show in Section 3.3.6 how Eve can discover a victim's cell number without the READ_PHONE_STATE permission.

Below we show how Eve can register with the UPI server as Alice, after Alice unwittingly installs Mally on her phone.

1. *Mally: I am installed!* Mally, once installed on Alice's phone, reports to Eve's C&C server over the Internet (Android automatically grants INTERNET permission). Mally reports Alice's cell number to Eve as a way for Eve (i) to discover Alice's cell number; and (ii) to associate the instance of Mally with Alice, which is essential for Eve to scale the attacks to many users.

2. *Eve: Use the cell number for hard binding:* Eve exploits *Potential Security Hole #1* in BHIM's workflow to bind her device to Alice's cell number as shown in Figure 4b. Eve starts by putting her cell phone in airplane mode while remaining connected to the Internet through Wi-Fi. BHIM app on Eve's phone starts the handshake by sending Eve's device details. The UPI server responds with a device registration token for Eve. Ideally, Eve's BHIM must relay the token back to the UPI server via SMS. However, since Eve has turned off SMS messaging, the SMS containing the token fails to deliver. BHIM prompts Eve to key-in a cell number and Eve keys-in Alice's cell number. BHIM now sends Eve's device registration token and Alice's cell number to the UPI server

as an HTTPS message for hard-binding. The UPI server then sends an OTP to Alice.

3. *Mally: Intercept the OTP.* On Alice's phone, Mally intercepts the incoming OTP message because its RECEIVE_SMS permission allows it. Mally then sends the OTP to the attacker's C&C server as an HTTPS message, along with Alice's cell number. (The cell number here is not strictly required. It merely allows the C&C server to associate each OTP with a victim and thus reduce some guesswork, in case it receives OTPs from other Mally installations.)

4. *Eve: Acknowledge the OTP.* The C&C server sends an SMS message containing the OTP to the attacker's phone. Note that the BHIM app normally checks the origin of the OTP message it receives and accepts the OTP only if it is from a known UPI server. However, Eve disabled this safeguard before the attack in the repackaged version of BHIM on her phone, thus exploiting *Potential Security Hole #2*

5. *Eve: New BHIM user? Create BHIM's Passcode:* BHIM on Eve's phone will ask for BHIM's 4-digit passcode. Now Eve does not know if Alice is a new user of BHIM or a registered user. However, Eve can determine this from Step 6 of the handshake where the UPI server sets a flag called *accountExists* to *false* for a new user. Eve can proceed to set a new passcode for a new user Alice. We discuss the workaround for the attack on an existing BHIM user in *Attack #1′*.

6. *Eve: Select the bank from the bank list.* Eve next selects each bank one-by-one on BHIM's bank selection screen until she finds one that the UPI server accepts. The UPI server will accept a bank if Alice has an account at that bank and has her cell number registered with that account.

   The UPI server does not appear to restrict brute-forcing—an error just brings the user back to the bank selection screen. In any case, brute-forcing is difficult to prevent since the list of banks is relatively short, and Eve can try out some of the larger banks where most people are likely to have an account with such as the State Bank of India or ICICI Bank.

Eve can repeat Attack #1 until she discovers all of Alice's bank accounts and registers with them.

3.3.3 Attack #1′: Eve: overcoming BHIM's passcode check for existing BHIM user

Attack #1 on a registered user Alice stalls when BHIM prompts the attacker Eve for Alice's BHIM passcode. We present three solutions to overcome the passcode barrier.

The first workaround is for Eve to wait for Mally to intercept and leak the new passcode. We found that Mally can do this as follows. *Mally* waits for Alice to launch BHIM. Mally detects BHIM's login activity to draw an overlay on it (see Figure 4c, keys demarcated for clarity). To draw the overlay, *Mally* exploits a toast overlay vulnerability CVE-2017-0752 [39] that requires no additional permissions from the user. Once Mally intercepts the passcode, it forwards the passcode to the C&C server.

The second workaround is for Mally to request and use Android's accessibility permission, which enables Mally to observe user interactions and intercept the passcode.

An attacker may, at this point, choose to reset the user's passcode. We find that BHIM's passcode reset workflow requires a user's bank account number instead of the debit card number. On the surface, it seems unlikely that Eve will know Alice's bank account number, and this, in isolation, may have been a reasonable passcode reset process. However, as described in *Potential Security Hole #3*, recall that the default UPI workflow reveals a user's bank account number. Eve can use the bank account number to reset Alice's BHIM passcode, courtesy of the UPI server.

Impact of Attack #1 and #1′. Eve cannot do transactions on the linked bank accounts after a successful registration. This attack, however, leaks private data such as the set of banks where Alice has bank accounts as well as Alice's bank account numbers. We also noticed that the UPI server sends a device registration token, a customer identifier, a login token, a hash of the account number, and the bank's account number back to BHIM (client) during the protocol handshake (see Figure 3). BHIM masks the bank account number but, nevertheless, the UPI server sends it, and Eve can get to it using the repackaged BHIM on Eve's phone. The Attack #1 is also a precursor to Attack #2 or Attack #3, which are more devastating. Note that the use of accessibility is only helpful in simplifying the attack; we do not require it for Attack #1.

3.3.4 Attack #2: Unauthorized transactions on bank accounts given cell number and partial debit card number

In this attack, which follows Attack #1, Eve extends the previous attack to enable transactions on a bank account of a user Alice that does not use any UPI apps. For the attack to succeed, Eve requires additional knowledge about Alice: the last six digits of Alice's debit card number and expiry date. Debit cards are carelessly given to unknown people in stores and restaurants in India at the time of checkout (often with cell numbers, as cashiers routinely collect cell numbers to send discount offers or give reward points). The majority of debit cards in India also carry the bank name. Using a debit card for purchases in stores or online in India requires the user to key-in a secret PIN. In this attack, even without the debit card PIN from Alice, with access to the debit card information

alone, Eve can set a UPI PIN to enable transactions on the associated bank account.

Impact. Losing or sharing one's debit card information along with the cell number (not the actual card, the actual cell, or the debit card PIN) can enable an attacker to set a UPI PIN and do transactions on one's bank account. Eve does not need bank account numbers or any of Alice's passcodes. The attack appears to be less scalable than Attack #1, however, since Eve needs to harvest debit card numbers along with associated cell numbers. For users who lose the two pieces of data to Eve and also install Mally, the impact is devastating. Eve could empty their account, with money transferred to any user in India. The attack does not even require a victim to have ever used a UPI app previously. To reset the UPI PIN, Eve requires the last six digits of the debit card number, expiry date, and an OTP, all of which she has.

3.3.5 Attack #3: Unauthorized transactions without debit card numbers

This attack follows from Attack #1′ for an existing user Alice. Such a user would have previously set up a passcode to log in to BHIM and UPI PIN to authorize transactions. Unfortunately, Mally can intercept the UPI PIN using either toast overlays or by requesting accessibility permission. As an alternative to intercepting UPI PIN, Eve can attempt to reset the UPI PIN (recall that Eve has already registered with the bank account in Attack #1′). As we described in the previous attack, resetting the UPI PIN requires debit card information, which reduces this attack to Attack #2. In short, either Mally intercepting UPI PIN or Eve possessing Alice's debit card information appears to be required. Eve now has all the factors to do transactions from her phone as Alice.

Impact: Eve can transfer money out to arbitrary UPI-based accounts in India. Note that for an attack on an existing user, Eve does not require any knowledge about Alice except for two things that Mally intercepts— an SMS message and the UPI PIN.

3.3.6 Eliminating the need for READ_PHONE_STATE permissions

The attacks we described so far relied on Mally knowing the victim's cell number and sending it to the C&C server, as a precursor to all the attacks. Now, we describe how Eve can associate a victim's cell number with an instance of Mally without Mally needing the READ_PHONE_STATE permission.

Given a set C of all targeted cell numbers (which is any list of cell numbers — valid or invalid), the following steps precede Attack #1:
(i) For each cell number in C, send an SMS to that number with the following content: [receiver's cell number, "SMS TEST"] (or any such message).
(ii) Consider a subset SC of phones C that have Mally installed. Mally looks for the string "SMS TEST" and saves the cell number in the SMS as the victim's cell number.
All instances of Mally that receive such an SMS message can thus learn their victim's cell number and report back to the C&C server to initiate the user registration protocol.

3.3.7 Whose problem: Android or UPI?

There is a potential question as to whether the attacks we discovered are primarily due to limitations of Android's permission model or due to flaws in the UPI design (and who should fix them). We think there are problems with both. We note that no bank-related credentials are required for an adversary to get a user's bank account number, given the user's cell number (in any of the handshakes– default or alternate). Attack #2 uses the last six digits of a debit card number and expiry date, a weaker threshold than for online and in-store purchases using debit cards where the entire number and the PIN is typically required in India. Alternate workflows in the UPI protocol contribute significantly to enabling our attacks. We, of course, leverage Android's security limitations as well, just as any good attacker would be expected to. We further discuss this issue in Section 5.

3.4 Other UPI 1.0 Apps

We now discuss whether the attacks on BHIM apply to the users of other UPI 1.0 apps. Our findings from testing three apps popular at the time of the study— PhonePe, Ola Wallet, and Samsung Pay—suggest yes. As shown in Figure 5, at the time of UPI 1.0, BHIM and PhonePe were the most popular UPI apps. PhonePe is also one of India's oldest payment apps. We did not include Google Pay (called Tez then) since it was not widely used, and Paytm was popular more for its wallet features. Below we discuss the attacks and its nuances under the same threat model.

First, these apps differ from BHIM because they are "third-parties" that integrate with UPI. Each third-party app uses its own factors for user profile setup. Hence, as discussed in the UPI specs Section 2.2, for third-party apps, their payment app server does the passcode-based authentication of a user while the UPI server verifies the device fingerprint and UPI PIN.

NPCI requires third-party apps to use NPCI's interface (libraries) for device fingerprinting and entering UPI PIN. We confirm that these apps internally use a common NPCI library to interface with the UPI server at the time of manual inspection. The UPI interface is accessible to a third-party app only after the user authenticates with the third-party payment app server. Thus, device binding and UPI PIN set up is done with the UPI server only after the user's passcode is set up with the payment app server.

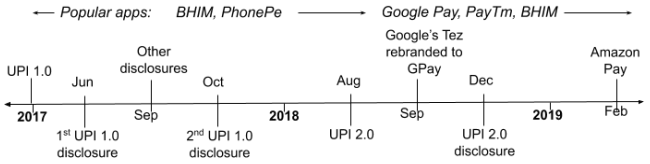


Figure 5: Popular UPI apps and disclosure timelines

Attack #1, unauthorized registration of a new user, can now be done by an adversary by setting up a user profile with the third-party app server and then exploiting the potential security holes of Section 3.3.1. Third-party apps make it easy for an attacker Eve to set up a profile. Eve can do it in two ways— Eve can either create a profile from her phone using her cell number (which is straight-forward) or create a profile from her phone using Alice's cell number. As an example of the latter, PhonePe provides an option to key-in a cell number at the time of user profile setup. Eve can use this option to key-in Alice's cell number in the app. For Eve to set a passcode on behalf of Alice, Eve needs an OTP the PhonePe server sends Alice. However, Eve can get the OTP through Mally on Alice's phone, given Mally's RECEIVE_SMS permission. The rest of Attack #1 continues as before, and Attack #2 follows from Attack #1.

For Attack #1′ on an existing user, an adversary can exploit any authentication workflow flaws on the third-party app or app server. Once logged in, Eve can exploit the potential security holes (Section 3.3.1). For Eve to log in as an existing user, Eve either has to get Alice's password or has to reset Alice's password. To get Alice's password, Mally can either use the toast overlay attacks or the accessibility permission. A straightforward approach, however, is to exploit the app's passcode reset mechanism. On PhonePe, for instance, the passcode reset relies only on an OTP. On Ola Money, passcode reset requires a secret that is set up at the time the user creates a profile (which we could intercept). We note that once Eve logs in as Alice on Eve's phone, PhonePe logs Alice out from her phone. In Ola Money, however, Alice will not receive any notification since the app by design permits login from many devices. The rest of Attack #1′ continues as before, and Attack #3 follows from Attack #1′.

Samsung Pay (SPay) is slightly different in that its security measures make use of a Trusted Execution Environment (TEE) [52] implementation called KNOX. To use SPay, a user must have a Samsung account configured at the time of setting up the phone and additionally configure her fingerprint or a SPay PIN. SPay does not integrate with UPI; instead, it integrates with two UPI apps—Paytm and MobiKwik. Hence a user can choose one of the two apps that come with SamsungPay (they are also available for download separately on Google Play). Since both Paytm and MobiKwik app servers do not integrate with KNOX, they cannot use KNOX's hardware-based security features for device hard-binding at the time of user registration. The user's fingerprint or SPay PIN is used to authenticate a user with the device; neither the payment app servers nor the UPI server uses it for user registration. We test SamsungPay using MobiKwik. Mobikiwk's workflow is the same as Ola Money except that its passcode reset workflow uses a passcode and OTP, both of which we can intercept. This makes SPay prone to attacks that result from integrating with third-party UPI apps.

3.5 UPI 1.0 Responsible Disclosures

We reported the vulnerabilities of BHIM to NPCI, CERT-IN, and CERT-US, with the initial disclosure to CERT-IN in June 2017. We followed up with our disclosures again in Oct 2017 (timelines in Figure 5). Subsequently, we reported the vulnerabilities to CERT-US and got the following CVEs: CVE-2017-9818, CVE-2017-9819, CVE-2017-9820, CVE-2017-9821 for BHIM. We also got CVEs for our disclosures to other app vendors from CERT-US (CVE-2018-15660, CVE-2018-15661, CVE-2018-17400, CVE-2018-17401, CVE-2018- 17402, CVE-2018-17403) and a $5k bounty from Samsung (CVE-2018-17083) for a sensitive data leak. The original CVEs disclosed relied on accessibility permission, though we later determined that the attacks can be carried out without it.

3.6 Preliminary Analysis of UPI 2.0 Protocol

In August 2018, UPI made the first update to the UPI specification, UPI 2.0, over a year after we first reported the vulnerabilities to them. Based on our disclosures, UPI 2.0 does prevent our attacks in the current form. We present our preliminary findings; a detailed analysis of UPI 2.0 is currently ongoing. We follow the same approach we employed for UPI 1.0 and reverse-engineered the UPI 2.0 protocol using UPI 2.0 versions of four popular apps— BHIM, Google Pay, Paytm, and Amazon Pay. Google Pay (GPay) and Paytm are the leaders in the market, each with a 36% market share.

Some of our findings are as follows. We evaluate the UPI 2.0 version of BHIM (which is also used by many banks as their official UPI app under their own brand, e.g., BHIM SBI Pay and BHIM PNB). We found that NPCI now forces an update on BHIM to its latest version. In UPI 2.0, in addition to the device information we saw in UPI 1.0, BHIM also sends the device's IMEI number, SIM number, network type, etc., to the UPI server for device hard-binding. In BHIM's latest update, NPCI removed the *alternate workflow1*, and hence the *Potential Security Hole #1* that we exploited for our attacks, a positive change. However, the other vulnerabilities persist as detailed below.

On GPay, we can set up a user's profile similar to how we did for Attack #1 and Attack #1′ in third-party UPI 1.0 apps. From GPay's traffic, we find that GPay authenticates with Gmail servers using OAuth2. Thus an adversary Eve can set up a GPay account as follows. Eve can use her own Gmail ID on her phone and can key-in Alice's cell number at the time of

login to GPay. Google sends an OTP to Alice's cell number, which Mally can intercept (given Mally's RECEIVE_SMS permission). For Eve to proceed, GPay must send an SMS message containing Alice's device registration token back to the UPI server from Alice's phone.

In the absence of the *alternate workflow1* that previously enabled the attacks, we explored SMS spoofing as a means for Eve to send an SMS message to the UPI server. For the attack to work, the UPI server must get the spoofed SMS message from Alice's cell number. For proof-of-concept, we tested SMS spoofing with several services that claim to provide non-anonymous SMS spoofing. However, it did not work for a test number we own in India. While we can send SMS messages either anonymously or using a default number provided by the SMS spoofing service, we are unable to control the sender number of the SMS message, a must for the attacks to work. We are currently exploring this and other SMS related attack vectors noted in prior research [49]. Alternatively, Mally can request SEND_SMS permission and send the SMS message from Alice's phone.

On Paytm, we studied the handshake by instrumenting the app with debug statements at the bytecode level. Below is a snippet of the bank account information that Paytm receives during the handshake. The authors mask all the details below for privacy. We note that just as before, UPI sends back the bank account details without requiring a user to provide any credentials shared with the bank. We confirm the same on Amazon Pay as well. Amazon Pay uses Amazon credentials and the default cell number set in a user's Amazon account. To create a profile, an adversary Eve can set Alice's cell number in her Amazon credentials.

```
"name":"956785XXXX@paytm",
"defaultCredit":{"bank":"State Bank Of India",
"ifsc":"SBIN0008626",
"account":"000000379085XXXXX",
"accountType":"SAVINGS",
"name":"BXXXXXX TXXXX",
"branchAddress":"AMXXXXXXXXX"
```

Thus, we have confirmed that sensitive information leaks (similar to those in Attack #1) still exist. An open question remains on the possibility of other attacks, such as performing unauthorized transactions.

4 Lessons Learned

Below, we summarize the problems in the design of the UPI 1.0 protocol that enabled potential attacks.

1. The UPI protocol reveals bank account details of a user in any handshake (default or alternate), given the user's cell number and no bank-related credentials.

2. Device hard-binding, the first factor, relies on data that is easily harvested from a device. UPI does not use any secrets for this step.

3. A weak device binding mechanism allows a user (or an adversary) to bind her cell phone with a cell number registered to the bank account of another user.

4. Setting the UPI PIN, the second factor, requires partial debit card information printed on the card, which is not a secret. The debit card PIN, a secret a user shares with the bank, is never used. This is a lower bar as online, and in-store purchases require the entire card number and the debit card PIN.

5. When transferring an existing user's UPI account to a new phone, UPI does not require the user to provide any bank-related credentials or the printed debit card information to authorize transactions from the new phone. The UPI protocol relies on the UPI PIN alone.

6. On third-party apps, the passcode, the third factor, is managed by the third-party app server and hence easy to bypass. An attacker can bypass the passcode requirement by setting up an attacker-controlled profile (using attacker credentials) with the app. In this case, UPI effectively relies only on two factors— device binding and UPI PIN.

7. The bank account number leaked from the default workflow of any of the third-party apps is enough to reset a user's passcode on another app (such as BHIM).

We note that though UPI 2.0 closes the weak device binding mechanism #3 above, the other issues persist. The overall weakness in UPI is that user registration requires only the knowledge of a cell number and the ability to receive one SMS message from that number.

Attacks only require Mally to do two things: provide the OTP during registration and, for attacks on existing users of UPI, steal their UPI PIN. Need for Mally can be circumvented in two ways— unauthorized transfer of a user's cell number to the attacker or by social engineering attacks. Both are feasible, and social engineering attacks are scalable in India, given the cheap labor cost. For non-users of UPI, getting them to reveal an OTP during registration is sufficient.

There are significant risks associated with relying on cell numbers as the only means of user identification. Banks in India accept any cell number that the user registers with their accounts—there is no cross-check to verify if the cell number given actually belongs to the user. It is not uncommon, for example, for members of a family to provide the same cell number to the bank for their individual bank accounts. Thus, a person with access to family members' debit card numbers can add all their bank accounts to the same app for transactions. One may view this either as a convenience or a security and privacy risk, depending on one's perspective.

Finally, we would like to clarify that our claim is not that all the high-level lessons learned are new; most security principles are well-known by now. Nevertheless, we want to con-

textualize the lessons learned from the perspective of a widely adopted financial protocol. We note that both the designers of Android and UPI contribute to the flaws we discovered, which made getting app vendors to do fixes difficult. App vendors often blame it on Android design or users, who should not be granting dangerous permissions to apps. At the same time, UPI protocol designers could have factored in the current state of Android and security-awareness among users in India and made the protocol more secure.

It is well-known by now that security by obscurity does not help. We think the risks could have been better addressed had UPI published the protocol details once it was internally vetted, thus allowing the research community to analyze it further. We show how protocol analysis from the point-of-view of an adversary trying to uncover unpublished workflows and secrets, though important, is often overlooked for application-level protocols.

Limitations of our study: A limitation of our study is that we only studied seven UPI apps to analyze the security of the UPI protocol. Automated analysis techniques could not be used given the number of security defenses these apps use. Prior research by Reaves et al. [48] also reverse-engineered seven apps that resisted automated analysis. However, we consider seven to be a reasonable number for our work since our focus was on uncovering flaws with the UPI protocol that is common across the apps. Also, the apps we analyzed have 88% of the market share combined, and of the 88 UPI apps, a majority of them are minor variations of BHIM, which we analyzed. Nevertheless, a larger study could provide additional insights into the security of the payment ecosystem in India and will also be useful to other countries that decide to use a common payment interface.

5 Mitigation

We discuss possible mitigation strategies against the attacks and their pros and cons below.

UPI mitigations. We discuss steps the government can take to address some of the issues we have raised.

Minimizing protocol data: Our attacks show how protocol data revealed during the default workflow was used to exploit an alternate workflow. This was possible because the UPI server sent more data than the client needed to see. For instance, while the masked bank account number is useful to display on the screen, bank-specific details such as the bank name, account number and IFSC code, sent in the clear can be excluded from the handshake.

Secure alternate workflows: We leveraged two alternate workflows in our attacks, as summarized in Section 4. Though UPI 2.0 closes one of the flows, the other alternate flows are either unsecured or secured using weak credentials. For instance, an alternate workflow allowed a user to bind her cell phone with a cell number registered to the bank account of another user, even without providing any secrets pertaining to the other user.

Mandate opt-in into UPI apps: Currently, as we are aware, UPI services are by default available to users of a bank that is integrated with UPI; the UPI guidelines do not require users to opt-in with their bank. An opt-in requirement would increase risk awareness as well as cut down security risks for non-UPI users such as credit card users, cash users, or users of wallet apps. Alternatively, a user could be required to do an in-person verification with their local bank branch to register for UPI services on their cell phone. This can prevent unauthorized registrations of a user, which automatically eliminates the other attacks.

Provide opt-out option: As a follow-up on the previous mitigation, non-users and users wanting to discontinue UPI services must be allowed to opt-out for security and privacy reasons. The downside of making UPI optional is the negative impact it may have on UPI adoption.

Use debit card number + something user knows: Debit cards in India are Chip+PIN cards, and doing transactions with them always requires entering a PIN. In contrast, doing transactions via the UPI apps requires neither—only the information that is printed on the card—resulting in a weaker authentication path. Fixes to this are unfortunately difficult if Mally is powerful enough to intercept PIN entry. However, assuming user interactions can be secured on Android (e.g., see [18]), UPI guidelines requiring the user to enter a secret shared with the bank to enable transactions will be useful.

Require strong device binding: The UPI specification could require payment apps to do a stronger device-to-cell number binding. Since binding is one of the most critical steps of the protocol, the bank may issue a one-time secret to the user out-of-band, say, when the user visits the bank for UPI activation. The user has to enter this secret the first time she uses the UPI app on her phone. Additionally, the UPI server must verify that the UPI app it is communicating with is an official app running on a non-rooted phone. If the UPI server can somehow establish that, then an attacker may not be able to use a repackaged version of a UPI app to register an account. Unfortunately, this is tricky to enforce.

Android mitigations. In the attacks we describe, the attack starts when Mally on a user's phone gets the user's cell number as an SMS from the attacker. A possible defense would be for Android to have a policy that prevents SMS permissions from being requested by apps. Google is already moving in that direction. As of January 2019, Google announced that apps could not request SMS permissions unless they are the default SMS handler and get explicit approval from Google. How effective this policy is, remains to be studied. We note that this does not make the attack impossible. It would merely require Mally not just to be installed but also accepted as

the default SMS handler (or get approved as an exception by Google). Also, the policy is specific to the Google Play store—apps from other stores could still introduce risks. Many popular carriers in India support alternate app stores such as Aircel and Airtel that allow SMS-triggered downloads [43].

User mitigations. Since Eve requires a user's cell number to initiate the attack, using a private cell number for bank accounts may slow down an attack. Unfortunately, it does not entirely prevent it. If the user has installed Mally, Mally suffices to detect the user's cell phone number (Section 3.3.6. Thus, users would also need to be careful to never install apps with read or receive SMS permissions on phones they use for banking.

6 Related Work

Panjwani et al. did one of the first studies on an Indian payment system called EKO, a mobile service provider [47]. They show PIN recovery attacks that could result in a user impersonation attack. Reaves et al. [48] first analyzed 47 mobile apps from 28 countries for SSL vulnerabilities and then manually reverse-engineer seven branchless banking apps, including three Indian payment apps (Airtel Money, Oxigen Wallet, and MobileOnMoney). They discover that an attacker can bypass authentication because of the use of an insecure channel, the use of weak crypto, or the use of weak passwords. A follow-up work by Castle et al [9] studies 197 payment apps, including some from Southern Asia (the apps they study is not listed). Castle et al. point out that payment apps have sufficient safeguards to prevent attacks, and the vulnerabilities pointed out by Reaves et al. are either because of regulatory constraints or from using old Android phones. They corroborate their findings with developer interviews with participants from well-established organizations.

Payment apps have been studied in other countries, as well. Yang et al. [62] notes implementation weaknesses in the third-party SDKs included by Chinese financial apps that can result in integrity attacks on financial transactions. Jung et al. [31] studies repackaging attacks on seven different banking apps in Korea. Their attacks could bypass integrity checks and anti-virus checks of banking apps. Yacouba et al. [33] launched a DDoS attack on a banking server through a repackaged banking app. Roland et al. demonstrates an NFC relay attack on the Google Wallet payment system [50].

Research has pointed out several vulnerabilities in financial applications. Taylor et al. [56] did a static analysis of financial apps on Google Play. They discover weaknesses such as the creation of world-readable and writable files, the use of unsecured content providers, and the use of weak random number generators. Bojjagani et al. [7] perform static and dynamic analysis on banking apps to discover 356 exploitable vulnerabilities, details unknown, from an unknown set of samples. AlJudaibi et al. [3] discuss 11 significant threats faced by mobile devices such as insecure data storage, weak server-side control in third-party apps, use of a rooted device, and lack of security in software and kernel. Chothia et al. [11], Stone et al. [55] and Bojjagani et al. [6] analyze both Android and iPhone apps for lack of hostname verification when an SSL certificate is pinned. Their results show how popular banking apps with these vulnerabilities are prone to phishing and man-in-the-middle attacks.

Protocol flaws that result in attacks on payment cards that use chip and PIN (EMV) [8, 35, 38, 51] and 3 Domain Secure 2.0 [2], an authentication protocol for web-based payments, are also studied before. Many issues concerning financial inclusion for developing countries such as Brazil and Africa have been extensively studied [22, 42, 61]. Weaknesses in financial systems as a result of excessive reliance on OTPs [10, 37, 49, 59] and its implication on Internet-based services are also well-known [1, 16, 32, 37, 53, 63].

Prior studies on Indian payments apps were done before the Indian government launched the Unified Payment Interface, a first of its kind. To the best of our knowledge, we are the first to conduct a study on UPI.

7 Conclusion

In this paper, we used a principled approach to analyze the UPI 1.0 protocol and uncovered core design weaknesses in its unpublished multi-factor authentication workflow that can severely impact a user. We showed attacks that have devastating implications and only require victims to have installed an attacker-controlled app, regardless of whether they use a UPI app or not. All the vulnerabilities identified were responsibly disclosed. A subsequent software update to UPI 2.0 prevents the discussed attack vectors for an exploit. Unfortunately, several underlying security flaws remain that suggest a need for further vetting and security analysis of UPI 2.0, given the protocol's importance for mobile payments in India. We discussed the lessons learned and potential mitigation strategies. Finally, we expect our findings to be useful to other countries that look to implement a common backend infrastructure for financial apps.

8 Acknowledgements

The authors thank Paul Pearce for shepherding the paper, the anonymous reviewers for their valuable inputs, and Roya Ensafi and Earlence Fernandes for their valuable feedback. We also thank colleagues, including Jithin M., Jothis M., and Arjun R., for helping us with analyzing Android apps in preliminary stages of the project when much less was known. This material is based on the work supported by the National Science Foundation under grant number 1646392.

References

[1] Manal Adham, Amir Azodi, Yvo Desmedt, and Ioannis Karaolis. How to attack two-factor authentication internet banking. In *Financial Cryptography and Data Security*, pages 322–328, 2013.

[2] Mohammed Aamir Ali and Aad van Moorsel. Designed to be broken: A reverse engineering study of the 3D Secure 2.0 Payment Protocol. In *Financial Cryptography and Data Security*, pages 201–221, 2019.

[3] Samaher AlJudaibi. Research paper for mobile devices security. 2016. https://www.researchgate.net/publication/309675787_Research_Paper_for_Mobile_Devices_Security.

[4] APKTOOL. https://ibotpeaches.github.io/Apktool/, 2018. [Online; accessed October-2018].

[5] BHIM. https://play.google.com/store/apps/details?id=in.org.npci.upiapp, 2016. [Online; accessed October-2018].

[6] S. Bojjagani and V. N. Sastry. VAPTAi: a threat model for vulnerability assessment and penetration testing of Android and iOS mobile banking apps. In *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pages 77–86, 10 2017.

[7] Sriramulu Bojjagani and V. N. Sastry. STAMBA: security testing for Android mobile banking apps. In *Advances in Signal Processing and Intelligent Recognition Systems*, pages 671–683, 2016.

[8] Mike Bond, Omar Choudary, Steven J. Murdoch, Sergei Skorobogatov, and Ross Anderson. Chip and Skim: Cloning EMV cards with the pre-play attack. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 49–64. IEEE Computer Society, 2014.

[9] Sam Castle, Fahad Pervaiz, Galen Weld, Franziska Roesner, and Richard Anderson. Let's talk money: Evaluating the security challenges of mobile money in the developing world. In *Proceedings of the 7th Annual Symposium on Computing for Development*, ACM DEV '16, 2016.

[10] Kelvin Chikomo, Ming Ki Chong, Alapan Arnab, and Andrew Hutchison. Security of mobile banking. 01 2006.

[11] Tom Chothia, Flavio D. Garcia, Chris Heppel, and Chris McMahon Stone. Why banker Bob (still) can't get TLS right: A security analysis of TLS in leading UK banking apps. pages 579–597, 01 2017.

[12] Express Computer, 2019. https://www.expresscomputer.in/news/financial-cybercrime-and-identity-theft-in-india-are-increasing-fis/35099/.

[13] Stat Counter. https://www.statista.com/statistics/262157/market-share-held-by-mobile-operating-systems-in-india/, 2018. [Online; accessed October-2018].

[14] CuckooDroid. https://github.com/idanr1986/cuckoo-droid), 2018. [Online; accessed October-2018].

[15] Android Developers. https://developer.android.com/guide/topics/permissions/overview#normal_permissions, 2019. [Online; accessed August-2019].

[16] Alexandra Dmitrienko, Christopher Liebchen, Christian Rossow, and Ahmad-Reza Sadeghi. On the (in)security of mobile two-factor authentication. In *Financial Cryptography and Data Security*, pages 365–383, 2014.

[17] Financial Express. https://www.financialexpress.com/money/beware-upi-app-user-loses-rs-6-8-lakh-from-his-sbi-account-was-it-fraud-why-it-happened/1426603/, 2018. [Online; accessed August-2019].

[18] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. Android UI deception revisited: Attacks and defenses. In *Financial Cryptography and Data Security*, pages 41–59, 2017.

[19] Forbes. https://www.forbes.com/sites/zakdoffman/2019/08/10/google-warning-tens-of-millions-of-android-phones-come-preloaded-with-dangerous-malware/#5dcde47dddb3, 2019. [Online; accessed August-2019].

[20] Sensors Tech Forum. https://sensorstechforum.com/android-ios-invasive-app-permissions-2018/), 2018. [Online; accessed October-2018].

[21] Mobile Security Framework. https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2015. [Online; accessed October-2018].

[22] Andrew Harris, Seymour Goodman, and Patrick Traynor. Privacy and security concerns associated with mobile money applications in Africa. *Washington Journal of Law, Technology and Arts*, 01 2013.

[23] The Hindu. https://www.thehindu.com/news/national/kerala/hackers-compromise-upi-apps/article25692100.ece, 2019. [Online; accessed August-2019].

[24] Android Security 2017 Year in Review. https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf, 2019. [Online; accessed August-2019].

[25] Android Security 2018 Year in Review. https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf/, 2019. [Online; accessed August-2019].

[26] MWR Infosecurity. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf, 2015. [Online; accessed October-2018].

[27] Business Insider. https://www.businessinsider.com/upi-market-share-in-inda-open-for-taking-2019-8, 2019. [Online; accessed August-2019].

[28] Infosec Institute. Hooking and patching Android apps using Xposed framework. https://resources.infosecinstitute.com/android-hacking-and-security-part-25-hooking-and-patching-android-apps-using-xposed-framework/#gref. [Online; accessed November-2019].

[29] Investopedia. https://www.investopedia.com/news/india-demonetization-993-money-returned/, 2018. [Online; accessed October-2018].

[30] JEB. https://www.pnfsoftware.com/, 2018. [Online; accessed October-2018].

[31] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. Repackaging attack on Android banking applications and its countermeasures. 73, 12 2013.

[32] Radhesh Krishnan Konoth, Victor van der Veen, and Herbert Bos. How anywhere computing just killed your phone-based two-factor authentication. In *Financial Cryptography and Data Security*, pages 405–421, 2017.

[33] Y. Kouraogo, K. Zkik, E. J. El Idrissi Noreddine, and G. Orhanou. Attacks on Android banking applications. In *2016 International Conference on Engineering MIS (ICEMIS)*, pages 1–6, 9 2016.

[34] The Hindu Business Line. https://www.thehindubusinessline.com/info-tech/senior-executives-vulnerable-to-social-engineering-attacks-verizon-2019-report/article27068079.ece, 2019. [Online; accessed August-2019].

[35] El Nour Madhoun, Bertin Emmanuel, and Guy Pujolle. The EMV payment system: Is it reliable? In *The 3rd IEEE Cyber Security in Networking International Conference (CSNet 2019)*, 2019.

[36] MITMProxy. https://mitmproxy.org, 2019. [Online; accessed August-2019].

[37] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. SMS-based One-Time Passwords: Attacks and defense. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 150–159, 2013.

[38] S. J. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and PIN is broken. In *2010 IEEE Symposium on Security and Privacy*, pages 433–446, 2010.

[39] Palo Alto Networks. https://researchcenter.paloaltonetworks.com/2017/09/unit42-android-toast-overlay-attack-cloak-and-dagger-with-no-permissions, 2017. [Online; accessed October-2018].

[40] NPCI. NPCI live members. https://www.npci.org.in/upi-live-members. [Online; accessed August-2019].

[41] NPCI. NPCI third party apps. https://www.npci.org.in/upi-PSP%263rdpartyApps. [Online; accessed August-2019].

[42] Barak W. Nyamtiga and Loserian S. Laizer. Enhanced security model for mobile banking systems in Tanzania. *International Journal of Technology Enhancements and Emerging Engineering Research*, 01 2013.

[43] Business of Apps. App stores list 2018. https://www.businessofapps.com/guide/app-stores-list/. [Online; accessed August-2019].

[44] National Payments Corporation of India. https://www.npci.org.in/sites/default/files/UPI-PG-RBI_Final.pdf, 2016. [Online; accessed October-2018].

[45] National Payments Corporation of India. https://www.npci.org.in/product-statistics/upi-product-statistics, 2019. [Online; accessed August-2019].

[46] OWASP. https://sushi2k.gitbooks.io/the-owasp-mobile-security-testing-guide/content/, 2018. [Online; accessed October-2018].

[47] Saurabh Panjwani and Edward Cutrell. Usably secure, low-cost authentication for mobile banking. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, Proc. SOUPS, 2010.

[48] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R.B. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *24th USENIX*

Security Symposium (USENIX Security 15), pages 17–32, 2015.

[49] Bradley Reaves, Luis Vargas, Nolen Scaife, Dave Tian, Logan Blue, Patrick Traynor, and Kevin R. B. Butler. Characterizing the security of the SMS ecosystem with public gateways. *ACM Trans. Priv. Secur.*, 22:2:1–2:31, December 2018.

[50] M. Roland, J. Langer, and J. Scharinger. Applying relay attacks to Google wallet. In *2013 5th International Workshop on Near Field Communication (NFC)*, pages 1–6, 2 2013.

[51] Michael Roland and Josef Langer. Cloning credit cards: A combined pre-play and downgrade attack on EMV contactless. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 6–6. USENIX Association, 2013.

[52] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 8 2015.

[53] Hossein Siadati, Toan Nguyen, Payas Gupta, Markus Jakobsson, and Nasir Memon. Mind your SMSes. *Comput. Secur.*, 65:14–28, March 2017.

[54] Soot. https://www.sable.mcgill.ca/soot/tutorial/profiler2/index.html, 2018. [Online; accessed October-2018].

[55] Chris McMahon Stone, Tom Chothia, and Flavio D. Garcia. Spinner: Semi-automatic detection of pinning without hostname verification. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 176–188, 2017.

[56] Vincent F. Taylor and Ivan Martinovic. Short paper: A longitudinal study of financial apps in the Google Play store. In *Financial Cryptography and Data Security*, pages 302–309, 2017.

[57] Economic Times. https://economictimes.indiatimes.com/news/politics-and-nation/new-form-of-otp-theft-on-rise-many-techies-victims/articleshow/67521098.cms, 2019. [Online; accessed August-2019].

[58] India Today. https://www.indiatoday.in/technology/news/story/fraudsters-steal-rs-91-000-from-a-man-s-e-wallet-1382689-2018-11-05, 2018. [Online; accessed August-2019].

[59] Patrick Traynor, Thomas La Porta, and Patrick McDaniel. *Security for telecommunications networks*. Advances in information security. 2008.

[60] Wandera. What are app permissions – a look into Android app permissions. https://www.wandera.com/mobile-security/app-and-data-leaks/app-permissions/. [Online; accessed August-2019].

[61] Jane K Winn and Louis De Koker. Introduction to mobile money in developing countries: Financial inclusion and financial integrity conference special issue. *University of Washington School of Law Research Paper*, (2013-01), 2013.

[62] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! Finding flawed implementations of third-party in-app payment in Android apps. In *NDSS*, 2017.

[63] Changsok Yoo, Byung-Tak Kang, and Huy Kang Kim. Case study of the vulnerability of OTP implemented in internet banking systems of South Korea. *Multimedia Tools and Applications*, 74:3289–3303, 05 2015.

9 Appendix

9.1 BHIM Code Instrumentation

We provide a brief discussion of one example instrumentation of BHIM with the goal of determining the workflow of the UPI protocol. BHIM version 1.3 consists of about 516K lines of obfuscated smali code. Some apps such as Paytm are even larger than BHIM, posing a significant reverse engineering challenge.

After searching through the BHIM code, we located the snippet below that belongs to the NPCI library and is integrated with the BHIM app. We found that NPCI had not obfuscated the name of the package as shown in line #1 *in/org/npci/upiapp/utils*. However, the method names are obfuscated as indicated by the method name at line #19 called *a*. The third-party libraries used by NPCI are not obfuscated as is seen by the class *org.apache.http.impl.client.DefaultHttpClient* at line #17.

We instrumented different portions of the BHIM app to determine the control-flow of the program. We found that when using automated tools such as Soot [54] to instrument the app, we got unexpected failures such as the app hanging indefinitely (we did get Soot to work for smaller test programs). We were unable to root-cause why BHIM's instrumentation with Soot did not work. Hence, we resorted to a careful smali code instrumentation of BHIM.

Listing 1 shows the method that performs HTTP GET. Since the methods are all *static* methods, by Android (and Java) convention, the first parameter is stored in the register *p0*, the second in register *p1* etc. The registers *v0*, *v1* etc. are registers local to a method body. Listing 2 contains code that prints the parameters to the GET request contained in the

parameter *p1*. We inserted the code in Listing 2 after line #38, right at the beginning of the function (after the function prologue at line #35). The inserted code snippet prints the parameters using the *System.out.print* API call. The printed debug statements appear in Android *logcat* logs. We did a similar instrumentation for HTTP POST methods.

Some of the apps such as Paytm, that contain several DEX files (with each DEX file containing a maximum of 65536 methods), were even more challenging to instrument, as they obfuscate the calls to most of the third-party libraries they use. In such cases, further experimentation and analysis was required to discover the calls. That apart, the security defenses used by these apps may also change across app revisions. For instance, while older versions of Paytm could be repackaged, the latest version of the app resists repackaging.

```
1  .class public Lin/org/npci/upiapp/utils/RestClient;
2  .super Ljava/lang/Object;
3  .source "RestClient.java"
4
5  # annotations
6  .annotation system Ldalvik/annotation/MemberClasses;
7      value = {
8          Lin/org/npci/upiapp/utils/
               RestClient$UnsuccessfulRestCall;
9      }
10 .end annotation
11
12 # static fields
13 .field private static final a:Ljava/lang/String;
14
15 .field private static b:Lorg/apache/http/impl/client/
       DefaultHttpClient;
16
17 .field private static c:Lorg/apache/http/impl/client/
       DefaultHttpClient;
18
19 .method public static a(Landroid/content/Context;Ljava/
       lang/String;Ljava/util/ Map;)Lin/org/npci/upiapp/
       models/ApiResponse;)
20     .locals 6
21     .annotation system Ldalvik/annotation/Signature;
22         value = {
23             "(",
24             "Landroid/content/Context;",
25             "Ljava/lang/String;",
26             "Ljava/util/Map",
27             "<",
28             "Ljava/lang/String;",
29             "Ljava/lang/String;",
30             ">;)",
31             "Lin/org/npci/upiapp/models/ApiResponse;"
32         }
33     .end annotation
34
35     .prologue
36     const/16 v5, 0x130
37
38     .line 404
39     new-instance v2, Lorg/apache/http/client/methods/
           HttpGet;
40
41     invoke-direct {v2}, Lorg/apache/http/client/methods/
           HttpGet;-><init>()V
42
43     .line 405
44     ...
45     move-result-object v2
46     const-string v3, " . Response Code:  "
47
48     invoke-virtual {v2, v3}, Ljava/lang/StringBuilder;->
           append(Ljava/lang/String;)Ljava/lang/
           StringBuilder;
49
50     move-result-object v2
51     invoke-interface {v0}, Lorg/apache/http/HttpResponse
           ;->getStatusLine()Lorg/apache/http/StatusLine;
52
53     move-result-object v0
54     invoke-interface {v0}, Lorg/apache/http/StatusLine;->
           getStatusCode()I
55
56     move-result v0
57     ...
58 .end method
```

Listing 1: BHIM code snippet

```
sget-object v0, Ljava/lang/System;->out:Ljava/io/
    PrintStream;

new-instance v1, Ljava/lang/StringBuilder;
invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

const-string/jumbo v2, "Log_debug_upi_str0: "
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {v1, p1}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString
    ()Ljava/lang/String;

move-result-object v1
invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(
    Ljava/lang/String;)V


sget-object v0,   Ljava/lang/System;->out:Ljava/io/
    PrintStream;

new-instance v1, Ljava/lang/StringBuilder;
invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

const-string/jumbo v2, "Log_debug_upi_restclient_map0: "
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {p2}, Ljava/lang/Object;->toString()Ljava/
    lang/String;

move-result-object v2
invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->
    append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v1
invoke-virtual {v1}, Ljava/lang/StringBuilder;->toString
    ()Ljava/lang/String;
move-result-object v1
invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(
    Ljava/lang/String;)V
```

Listing 2: HTTP GET Instrumentation Code