# Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World

Bradley Reaves
*University of Florida*
*reaves@ufl.edu*

Nolen Scaife
*University of Florida*
*scaife@ufl.edu*

Adam Bates
*University of Florida*
*adammbates@ufl.edu*

Patrick Traynor
*University of Florida*
*traynor@cise.ufl.edu*

Kevin R.B. Butler
*University of Florida*
*butler@ufl.edu*

## Abstract

Mobile money, also known as branchless banking, brings much-needed financial services to the unbanked in the developing world. Leveraging ubiquitous cellular networks, these services are now being deployed as smart phone apps, providing an electronic payment infrastructure where alternatives such as credit cards generally do not exist. Although widely marketed as a more secure option to cash, these applications are often not subject to the traditional regulations applied in the financial sector, leaving doubt as to the veracity of such claims. In this paper, we evaluate these claims and perform the first in-depth measurement analysis of branchless banking applications. We first perform an automated analysis of all 46 known Android mobile money apps across the 246 known mobile money providers and demonstrate that automated analysis fails to provide reliable insights. We subsequently perform comprehensive manual teardown of the registration, login, and transaction procedures of a diverse 15% of these apps. We uncover pervasive and systemic vulnerabilities spanning botched certification validation, do-it-yourself cryptography, and myriad other forms of information leakage that allow an attacker to impersonate legitimate users, modify transactions in flight, and steal financial records. These findings confirm that the majority of these apps fail to provide the protections needed by financial services. Finally, through inspection of providers' terms of service, we also discover that liability for these problems unfairly rests on the shoulders of the customer, threatening to erode trust in branchless banking and hinder efforts for global financial inclusion.

## 1 Introduction

The majority of modern commerce relies on cashless payment systems. Developed economies depend on the near instantaneous movement of money, often across great distances, in order to fuel the engines of industry. These rapid, regular, and massive exchanges have created significant opportunities for employment and progress, propelling forward growth and prosperity in participating countries. Unfortunately, not all economies have access to the benefits of such systems and throughout much of the developing world, physical currency remains the de facto means of exchange.

Mobile money, also known as branchless banking, applications attempt to fill this void. Generally deployed by companies outside of the traditional financial services sector (e.g., telecommunications providers), branchless banking systems rely on the near ubiquitous deployment of cellular networks and mobile devices around the world. Customers can not only deposit their physical currency through a range of independent vendors, but can also perform direct peer-to-peer payments and convert credits from such transactions back into cash. Over the past decade, these systems have helped to raise the standard of living and have revolutionized the way in which money is used in developing economies. Over 30% of the GDP in many such nations can now be attributed to branchless banking applications [39], many of which now perform more transactions per month than traditional payment processors, including PayPal [36].

One of the biggest perceived advantages of these applications is security. Whereas carrying large amounts of currency long distances can be dangerous to physical security, branchless banking applications can allow for commercial transactions to occur without the risk of theft. Accordingly, these systems are marketed as a secure new means of enabling commerce. Unfortunately, the strength of such claims from a *technical* perspective has not been publicly investigated or verified. Such an analysis is therefore critical to the continued growth of branchless banking systems.

In this paper, we perform the first comprehensive analysis of branchless banking applications. Through these efforts, we make the following contributions:

- **Analysis of Branchless Banking Applications:** We perform the first comprehensive security analysis of branchless banking applications. First, we use a well-known automated analysis tool on all 46 known Android mobile money apps across all 246 known mobile money systems. We then methodically select seven Android-based branchless banking applications from Brazil, India, Indonesia, Thailand, and the Phillipines with a combined user base of millions. We then develop and execute a comprehensive, reproducible methodology for analyzing the entire application communication flow. In so doing, we create the first snapshot of the global state of security for such applications.

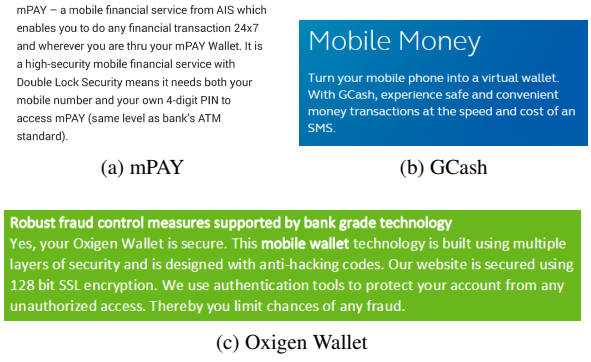
- **Identifications of Systemic Vulnerabilities:** Our analysis discovers pervasive weaknesses and shows that six of the seven applications broadly fail to preserve the integrity of their transactions. We then compare our results to those provided through automated analysis and show that current tools do not reliably indicate severe, systemic security faults. Accordingly, neither users nor providers can reason about the veracity of requests by the majority of these systems.

- **Analysis of Liability:** We combine our technical findings with the assignment of liability described within every application's terms of service, and determine that users of these applications have no recourse for fraudulent activity. Therefore, it is our belief that these applications create significant financial dangers for their users.

The remainder of this paper is organized as follows: Section 2 provides background information on branchless banking and describes how these applications compare to other mobile payment systems; Section 3 details our methodology and analysis architecture; Section 4 presents our findings and categorizes them in terms of the CWE classification system; Section 5 delivers discussion and recommendations for technical remediation; Section 6 offers an analysis of the Terms of Service and the assignment of liability; Section 7 discusses relevant related work; and Section 8 provides concluding remarks.

## 2 Mobile Money in the Developing World

The lack of access to basic financial services is a contributing factor to poverty throughout the world [17]. Millions live without access to basic banking services, such as value storage and electronic payments, simply because they live hours or days away from the nearest bank branch. Lacking this access makes it more difficult for the poor to save for future goals or prepare for future




(a) mPAY (b) GCash

(c) Oxigen Wallet

Figure 1: Mobile money apps are heavily marketed as being safe to use. These screenshots from providers' marketing materials show the extent of these claims.

setbacks, conduct commerce remotely, or protect money against loss or theft. Accordingly, providing banking through mobile phone networks offers the promise of dramatically improving the lives of the world's poor.

The M-Pesa system in Kenya [21] pioneered the *mobile money* service model, in which agents (typically local shopkeepers) act as intermediaries for deposits, withdrawals, and sometimes registration. Both agents and users interact with the mobile money system using SMS or a special application menu enabled by code on a SIM card, enabling users to send money, make bill payments, top up airtime, and check account balances. The key feature of M-Pesa and other systems is that their use does not require having a previously established relationship with a bank. In effect, mobile money systems are bootstrapping an alternative banking infrastructure in areas where traditional banking is impractical, uneconomic due to minimum balances, or simply non-existent. M-Pesa has not yet released a mobile app, but is arguably the most impactful mobile money system and highlights the promise of branchless banking for developing economies.

Mobile money has become ubiquitous and essential. M-Pesa boasts more than 18.2 million registrations in a country of 43.2 million [37]. In Kenya and eight other countries, there are more mobile money accounts than bank accounts. As of August 2014, there were a total of 246 mobile money services in 88 countries serving a total of over 203 million registered accounts, continuing a trend [49] up from 219 services in December 2013. Note that these numbers explicitly exclude services that are simply a mobile interface for existing banking systems. Financial security, and trust in branchless banking systems, depends on the assurances that these systems are secure against fraud and attack. Several of the apps that we study offer strong assurances of security in their promotional materials. Figure 1 provides examples

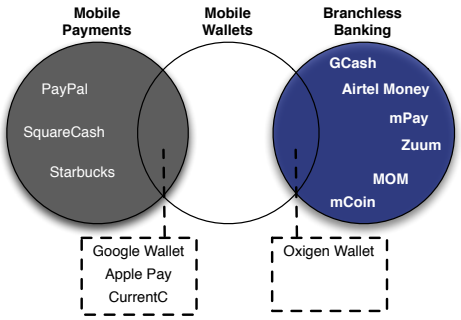



Figure 2: While Mobile Money (Branchless Banking) and Mobile Payments share significant overlapping functionality, the key differences are the communications methods the systems use and that mobile money systems do not rely on existing banking infrastructure.

of these promises. This promise of financial security is even reflected in the M-Pesa advertising slogan "Relax, you have got M-Pesa." [52]. However, the veracity of these claims is unknown.

## 2.1 Comparison to Other Services

Mobile money is closely related to other technologies. Figure 2 presents a Venn diagram indicating how representative mobile apps fall into the categories of mobile payments, mobile wallets, and branchless banking systems. Most mobile finance systems share the ability to make payments to other individuals or merchants. In our study, the mobile apps for these finance systems are distinguished as follows:

- **Mobile Payment** describes systems that allow a mobile device to make a payment to an individual or merchant using *traditional banking infrastructure*. Example systems include PayPal, Google Wallet, Apple Pay, Softpay (formerly ISIS), CurrentC, and Square Cash. These systems acting as an intermediary for an existing credit card or bank account.

- **Mobile Wallets** store multiple payment credentials for either mobile money or mobile payment systems and/or facilitate promotional offers, discounts, or loyalty programs. Many mobile money systems (like Oxigen Wallet, analyzed in this paper) and mobile payment systems (like Google Wallet and Apple Pay) are also mobile wallets.

- **Branchless Banking** is designed around policies that facilitate easy inclusion. Enrollment often simply requires just a phone number or national ID number be entered into the mobile money system. These systems have no minimum balances and low transaction fees, and feature reduced "Know Your Customer"[1] regulations [51]. Another key feature of branchless banking systems is that in many cases they do not rely on Internet (IP) connectivity exclusively, but also use SMS, Unstructured Supplementary Service Data (USSD), or cellular voice (via Interactive Voice Response, or IVR) to conduct transactions. While methods for protecting data confidentiality and integrity over IP are well established, the channel cryptography used for USSD and SMS has been known to be vulnerable for quite some time [56].

# 3 App Selection and Analysis

In this section, we discuss how apps were chosen for analysis and how the analysis was conducted.

## 3.1 Mallodroid Analysis

Using data from the GSMA Mobile Money Tracker [6], we identified a total of 47 Android mobile money apps across 28 countries. We first ran an automated analysis on all 47 of these apps using Mallodroid [28], a static analysis tool for detecting SSL/TLS vulnerabilities, in order to establish a baseline. Table 3 in the appendix provides a comprehensive list of the known Android mobile money applications and their static analysis results. Mallodroid detects vulnerabilities in 24 apps, but its analysis only provides a basic indicator of problematic code; it does not, as we show, exclude dead code or detect major flaws in design. For example, it cannot guarantee that sensitive flows *actually use* SSL/TLS. It similarly cannot detect ecosystem vulnerabilities, including the use of deprecated, vulnerable, or incorrect SSL/TLS configurations on remote servers. Finally, the absence of SSL/TLS does not necessarily condemn an application, as applications can still operate securely using other protocols. Accordingly, such automated analysis provides an incomplete picture at best, and at worst an incorrect one. This is a limitation of all automatic approaches, not just Mallodroid.

In the original Mallodroid paper, its authors performed a manual analysis on 100 of the 1,074 (9.3%) apps their tool detected to verify its findings; however, only 41% of those apps were vulnerable to SSL/TLS man-in-the-middle attacks. It is therefore imperative to further verify the findings of this tool to remove false positives and false negatives.

[1] "Know Your Customer" (KYC), "Anti-Money Laundering" (AML), and "Combating Financing of Terrorism" policies are regulations used throughout the world to frustrate financial crime activity.

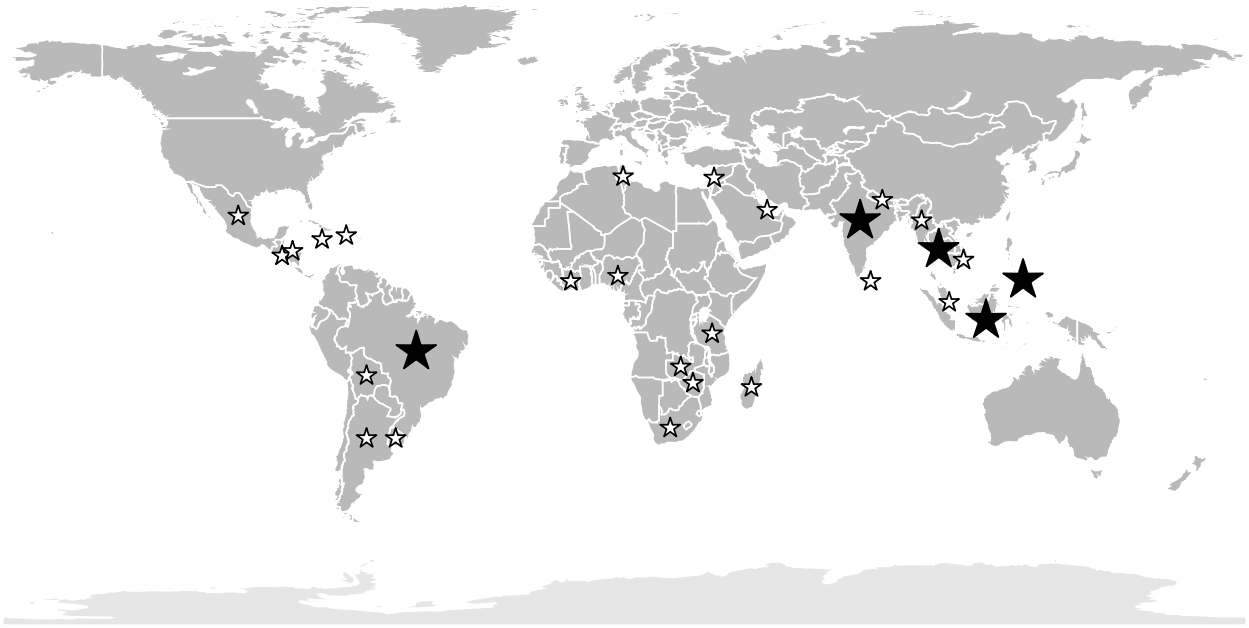


Figure 3: The mobile money applications we analyzed were developed for a diverse range of countries. In total, we performed an initial analysis on applications from 28 countries representing up to approximately 1.2 million users based on market download counts. From this, we selected 7 applications to fully analyze from 5 countries. Each black star represents these countries, and the white stars represent the remainder of the mobile money systems.

## 3.2 App Selection

Given the above observations, we selected seven mobile money apps for more extensive analysis. These seven apps represent 15% of the total applications and were selected to reflect diversity across markets, providers, features, download counts, and static analysis results. Collectively, these apps serve millions of users. Figure 3 shows the geographic diversity across all of the mobile money apps we observed and those we selected for manual analysis.

We focus on Android applications in this paper because Android has the largest market share worldwide [43], and far more mobile money applications are available for Android than iOS. However, while we cannot make claims about iOS apps that we did not analyze, we do note that all errors disclosed in Section 4 are possible in iOS and are not specific to Android.

## 3.3 Manual Analysis Process

Our analysis is the first of its kind to perform an in-depth analysis of the protocols used by these applications as well as inspect *both ends* of the SSL/TLS sessions they may use. Each layer of the communication builds on the last; any error in implementation potentially affects the security guarantees of all others. This holistic view of the entire app communication protocol at multiple layers offers a deep view of the fragility of these systems.

In order to accomplish this, our analysis consisted of two phases. The first phase provided an overview of the functionality provided by the app; this included analyzing the application's code and manifest and testing the quality of any SSL/TLS implementations on remote servers. Where possible, we obtained an in-country phone number and created an account for the mobile money system. The overview phase was followed by a reverse engineering phase involving manual analysis of the code. For those apps that we possessed accounts, we also executed the app and verified any findings we found in the code.

Our main interest is in verifying the integrity of these sensitive financial apps. While privacy issues like IMEI or location leakage are concerning [26], we focus on communications between the app and the IP or SMS backend systems, where an adversary can observe, modify, and/or generate transactions.

**Phase 1: Inspection**

In the inspection phase, we determined the basic functionality and structure of the app in order to guide later analysis. Figure 4 shows the overall toolchain for analyzing the apps along with each respective output.

The first step of the analysis was to obtain the application manifest using `apktool` [2]. We then used an simple script to generate a report identifying each app component (i.e., activities, services, content providers, and

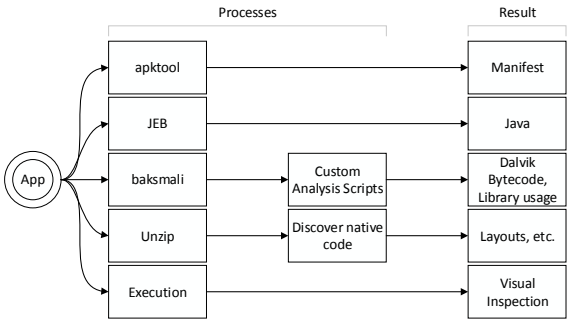



Figure 4: A visualization of the tools used for analyzing the mobile money apps.

broadcast receivers) as well as the permissions declared and defined by the app. This acted as a high-level proxy for understanding the capabilities of the app. With this report, we could note interesting or dangerous permissions (e.g., WRITE_EXTERNAL_STORAGE can leak sensitive information) and which activities are exported to other apps (which can be used to bypass authentication).

The second step of this phase was an automated review of the Dalvik bytecode. We used the `Baksmali` [10] tool to disassemble the application `dex` file. While disassembly provides the Dalvik bytecode, this alone does not assist in reasoning about the protocols, data flows, and behavior of an application. Further inspection is still required to understand the semantic context and interactions of classes and functions.

After obtaining the Dalvik bytecode, we used a script to identify classes that use interesting libraries; these included networking libraries, cryptography libraries (including `java.security` and `javax.crypto` and Bouncy Castle [11]), well-known advertising libraries (as identified by Chen et al. [18]), and libraries that can be used to evade security analysis (like Java ClassLoaders). References to these libraries are found directly in the Dalvik assembly with regular expressions. The third step of the overview was to manually take note of all packages included in the app (external libraries like social media libraries, user interface code, HTTP libraries, etc.).

While analyzing the application's code can provide deep insight into the application's behavior and client/server protocols, it does not provide any indication of the security of the connection as it is negotiated by the server. For example, SSL/TLS servers can offer vulnerable versions of the protocol, weak signature algorithms, and/or expired or invalid certificates. Therefore, the final step of the analysis was to check each application's SSL endpoints using the Qualys SSL Server Test [50].[2] This test provides a comprehensive, non-invasive view of the configuration and capabilities of a server's SSL/TLS implementation.

**Phase 2: Reverse Engineering**

In order to complete our holistic view of both the application protocols and the client/server SSL/TLS negotiation, we reverse engineered each app in the second phase. For this step, we used the commercial interactive JEB Decompiler [4] to provide Java syntax for most classes. While we primarily used the decompiled output for analysis, we also reviewed the Dalvik assembly to find vulnerabilities. Where we were able to obtain accounts for mobile money accounts, we also confirmed each vulnerability with our accounts when doing so would not negatively impact the service or other users.

Rather than start by identifying interesting methods and classes, we began analysis by following the application lifecycle as the Android framework does, starting with the `Application.onCreate()` method and moving on to the first Activity to execute. From there, we constructed the possible control paths a user can take from the beginning through account registration, login, and money transfer. This approach ensures that our findings are actually present in live code, and accordingly leads to conservative claims about vulnerabilities.[3] After tracing control paths through the `Activity` user interface code, we also analyze other components that appear to have sensitive functionality.

As stated previously, our main interest is in verifying the integrity of these financial applications. In the course of analysis, we look for security errors in the following actions:

- Registration and login
- User authentication after login
- Money transfers

These errors can be classified as:

- Improper authentication procedures
- Message confidentiality and integrity failures (including misuse of cryptography)
- Highly sensitive information leakage (including financial information or authentication credentials)
- Practices that discourage good security hygiene, such as permitting insecure passwords

We discuss our specific findings in Section 4.

### 3.3.1 Vulnerability Disclosure

As of the publication deadline of this paper we have notified all services of the vulnerabilities. We also included basic details of accepted mitigating practices for each

[2]For security reasons, Qualys does not test application endpoints on non-standard ports or without registered domain names.

[3]In the course of analysis, we found several vulnerabilities in what is apparently dead code. While we disclosed those findings to developers for completeness, we omit them from this paper.

| ID | Common Weakness Enumeration | Airtel Money | mPAY | Oxigen Wallet | GCash | Zuum | MOM | mCoin |
|---|---|---|---|---|---|---|---|---|
| *SSL/TLS & Certificate Verification* | | | | | | | | |
| CWE-295 | Improper Certificate Validation | ✗ | ✗ | | ✗ | | | ✗ |
| *Non-standard Cryptography* | | | | | | | | |
| CWE-330 | Use of Insufficiently Random Values | ✗ | | ✗ | ✗ | | | |
| CWE-322 | Key Exchange without Entity Authentication | | | ✗ | | | ✗ | |
| *Access Control* | | | | | | | | |
| CWE-88 | Argument Injection or Modification | | ✗ | | | | | |
| CWE-302 | Authentication Bypass by Assumed-Immutable Data | | | ✗ | ✗ | | | |
| CWE-521 | Weak Password Requirements | | | | ✗ | | | |
| CWE-522 | Insufficiently Protected Credentials | | | | | | ✗ | |
| CWE-603 | Use of Client-Side Authentication | | | | | | ✗ | |
| CWE-640 | Weak Password Recovery Mechanism for Forgotten Password | | | ✗ | | | | |
| *Information Leakage* | | | | | | | | |
| CWE-200 | Information Exposure | | | ✗ | | | ✗ | ✗ |
| CWE-532 | Information Exposure Through Log Files | | ✗ | | ✗ | | ✗ | |
| CWE-312 | Cleartext Storage of Sensitive Information | | ✗ | | ✗ | | | ✗ |
| CWE-319 | Cleartext Transmission of Sensitive Information | | | ✗ | ✗ | | ✗ | |

Table 1: Weaknesses in Mobile Money Applications, indexed to corresponding Common Weakness Enumeration (CWE) records. The CWE database is a comprehensive taxonomy of software vulnerabilities developed by MITRE [55] and provide a common language for software errors.

finding. Most have not sent any response to our disclosures. We have chosen to publicly disclose these vulnerabilities in this paper out of an obligation to inform users of the risks they face in using these insecure services.

# 4 Results

This section details the results of analyzing the mobile money applications. Overall, we find 28 significant vulnerabilities across seven applications. Table 1 shows these vulnerabilities indexed by CWE and broad categories (apps are ordered by download count). All but one application (Zuum) presents at least one major vulnerability that harmed the confidentiality of user financial information or the integrity of transactions, and most applications have difficulty with the proper use of cryptography in some form.

## 4.1 Automated Analysis

Our results for SSL/TLS vulnerabilities should mirror the output of an SSL/TLS vulnerability scanner such as Mallodroid. Though two applications were unable to be analyzed by Mallodroid, it detects at least one critical vulnerability in over 50% of the applications it successfully completed.

Mallodroid produces a false positive when it detects an SSL/TLS vulnerability in Zuum, an application that, through manual analysis, we verified was correctly performing certificate validation. The Zuum application *does contain* disabled certificate validation routines, but these are correctly enclosed in logic that checks for development modes.

Conversely, in the case of MoneyOnMobile, Mallodroid produces a false negative. MoneyOnMobile contains no SSL/TLS vulnerability because it does not employ SSL/TLS. While this can be considered correct operation of Mallodroid, it also does not capture the severe information exposure vulnerability in the app.

Overall, we find that Mallodroid, an extremely popular analysis tool for Android apps, does not detect the *correct* use of SSL/TLS in an application. It produces an alert for the most secure app we analyzed and did not for the least. In both cases, manual analysis reveals stark differences between the Mallodroid results and the real security of an app. A comprehensive, correct analysis must include a review of the application's validation and actual use of SSL/TLS sessions as well as *where* these are used in the application (e.g., used for all sensitive communications). Additionally, it is critical to understand whether the remote server enforces secure protocol versions, ciphers, and hashing algorithms. Only a manual analysis provides this holistic view of the communication between application and server so that a complete security evaluation can be made.

## 4.2 SSL/TLS

As we discussed above, problems with SSL/TLS certificate validation represented the most common vulnerability we found among apps we analyzed. Certificate validation methods inspect a received certificate to ensure that it matches the host in the URL, that it has a trust chain that terminates in a trusted certificate authority, and that it has not been revoked or expired. However, developers are able to disable this validation by creating a new class that implements the `X509TrustManager` interface using arbitrary validation methods, replacing the validation implemented in the parent library. In the applications that override the default code, the routines were empty; that is, they *do nothing* and do not throw excep-

| Product | Qualys Score | Most Noteworthy Vulnerabilities |
|---|---|---|
| Airtel Money | A- | Weak signature algorithm (SHA1withRSA) |
| mPAY 1 | F | SSL2 support, Insecure Client-Initiated Renegot. |
| mPAY 2 | F | Vulnerable to POODLE attack |
| Oxigen Wallet | F | SSL2 support, MD5 cipher suite |
| Zuum | A- | Weak signature algorithm (SHA1withRSA) |
| GCash | C | Vulnerable to POODLE attack |
| mCoin | N/A | Uses expired, `localhost` self-signed certificate |
| MoneyOnMobile | N/A | App does not use SSL/TLS |

Table 2: Qualys reports for domains associated with branchless banking apps. "Most Noteworthy Vulnerabilities" lists what Qualys considers to be the most dangerous elements of the server's configuration. mPAY contacts two domains over SSL, both of which are separately tabulated below. Qualys would not scan mCoin because it connects to a specific IP address, not a domain.

tions on invalid certificates. This insecure practice was previously identified by Georgiev et al. [31] and is specifically targeted by Mallodroid.

Analyzing only the app does not provide complete visibility to the overall security state of an SSL/TLS session. Server misconfiguration can introduce additional vulnerabilities, even when the client application uses correctly implemented SSL/TLS. To account for this, we also ran the Qualys SSL Server Test [50] on each of the HTTPS endpoints we discovered while analyzing the apps. This service tests a number of properties of each server to identify configuration and implementation errors and provide a "grade" for the configuration. These results are presented in Table 2. Three of the endpoints we tested received failing scores due to insecure implementations of SSL/TLS. To underscore the severity of these misconfigurations, we have included the "Most Noteworthy Vulnerabilities" identified by Qualys.

**mCoin.** Coupling the manual analysis with the Qualys results, we found that in one case, the disabled validation routines were required for the application to function correctly. The mCoin API server provides a certificate that is issued to "localhost" (an invalid hostname for an external service), is expired, and is self-signed (has no trust chain). *No correct certificate validation routine would accept this certificate*. Therefore, without this routine, the mCoin application would be unable to establish a connection to its server. Although Mallodroid detected the disabled validation routines, only our full analysis can detect the relationship between the app's behavior and the server's configuration.

The implications of poor validation practices are severe, especially in these critical financial applications. Adversaries can intercept this traffic and sniff cleartext personal or financial information. Furthermore, without additional message integrity checking inside these weak SSL/TLS sessions, a man-in-the-middle adversary is free to manipulate the inside messages.

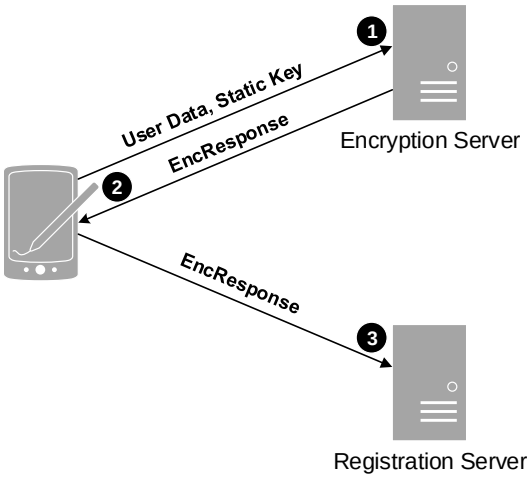



Figure 5: The user registration flow of MoneyOnMobile. All communication is over HTTP.

## 4.3 Non-Standard Cryptography

Despite the pervasive insecure implementations of SSL/TLS, the client/server protocols that these apps implement are similarly critical to their overall security. We found that four applications used their own custom cryptographic systems or had poor implementations of well-known systems in their protocols. Unfortunately, these practices are easily compromised and severely limit the integrity and privacy guarantees of the software, giving rise to the threat of forged transactions and loss of transaction privacy.

**MoneyOnMobile.** MoneyOnMobile does not use SSL/TLS. All API calls from the app use HTTP. In fact, we found only one use of cryptography in the application's network calls. During the user registration process, the app first calls an encryption proxy web service, then sends the service's response to a registration web service. The call to the encryption server includes both the user data and a fixed static key. A visualization of this protocol is shown in Figure 5.

The encryption server is accessed over the Internet via HTTP, exposing both the user and key data. Because this data is exposed during the initial call, its subsequent encryption and delivery to the registration service provides no security. We found no other uses of this or any other encryption in the MoneyOnMobile app; all other API calls are provided unobfuscated user data as input.

**Oxigen Wallet.** Like MoneyOnMobile, Oxigen Wallet does not use SSL/TLS. Oxigen Wallet's registration messages are instead encrypted using the Blowfish algorithm, a strong block cipher. However, a long, random key is not generated for input into Blowfish. In-

stead, only 17 bits of the key are random. The remaining bits are filled by the mobile phone number, the date, and padding with 0s. The random bits are generated by the `Random` [34] random number generator. The standard Java documentation [44] *explicitly warns* in its documentation that `Random` is not sufficiently random for cryptographic key generation.[4] As a result, any attacker can read, modify, or spoof messages. These messages contain demographic information including first and last name, email address, date of birth, and mobile phone number, which constitutes a privacy concern for Oxigen Wallet's users.

After key generation, Oxigen Wallet transmits the key in plaintext along with the message to the server. In other words, every encrypted registration message *includes the key in plaintext*. Naturally, this voids every guarantee of the block cipher. In fact, any attacker who can listen to messages can decrypt and modify them with only a few lines of code.

The remainder of client-server interactions use an RSA public key to send messages to the server. To establish an RSA key for the server, Oxigen Wallet sends a simple HTTP request to receive an RSA key from the Oxigen Wallet server. This message is unauthenticated, which prevents the application from knowing that the received key is from Oxigen Wallet and not from an attacker. Thus, an attacker can pretend to be Oxigen Wallet and send an alternate key to the app. This would allow the attacker to read all messages sent by the client (including those containing passwords) and forward the messages to Oxigen Wallet (with or without modifications) if desired. This RSA man-in-the-middle attack is severe and puts all transactions by a user at risk. At the very least, this will allow an attacker to steal the password from messages. The password can later be used to conduct illicit transactions from the victim's account.

Finally, responses from the Oxigen Wallet servers are not encrypted. This means that any sensitive information that might be contained in a response (e.g., the name of a transaction recipient) can be read by any eavesdropper. This is both a privacy and integrity concern because an attacker could read and modify responses.

**GCash.** Unlike Oxigen Wallet, GCash uses a static key for encrypting communcations with the remote server. The GCash application package includes a file "`enc.key`," which contains a symmetric key. During the GCash login process, the user's PIN and session ID are encrypted using this key before being sent to the GCash servers. This key is posted publicly because it is included with every download of GCash. An attacker with this key can decrypt the user's PIN and session ID if the encrypted data is captured. This can subsequently give the attacker the ability to impersonate the user.

The session ID described above is generated during the login process and passed to the server to provide session authentication in subsequent messages. We did not find any other authenticator passed in the message body to the GCash servers after login. The session ID is created using a combination of the device ID, e.g., International Mobile Station Equipment Identity (IMEI), and the device's current date and time. Android will provide this device ID to any application with the READ_PHONE_STATE permission, and device IDs can be spoofed on rooted phones. Additionally, IMEI is frequently abused by mobile apps for persistent tracking of users [25], and is thus also stored in the databases of hundreds of products.

Although the session ID is not a cryptographic construct, the randomness properties required by a strong session ID match those needed by a strong cryptographic key. This lack of randomness results in predictable session IDs can then be used to perform any task as the session's associated user.

**Airtel Money.** Airtel Money performs a similar mistake while authenticating the user. When launching the application, the client first sends the device's phone number to check if there is an existing Airtel Money account. If so, the server sends back the user's account number in its response. Although this response is transmitted via HTTPS, the app does not validate certificates, creating a compound vulnerability where this information can be discovered by an attacker.

Sensitive operations are secured by the user's 4-digit PIN. The PIN is encrypted in transit using a weakly-constructed key that concatenates the device's phone number and account number in the following format:

$$Key_{enc} = \texttt{j7zgy1yv} \,\|\, phone\# \,\|\, account\# \qquad (1)$$

The prefixed text in the key is immutable and included with the application. Due to the weak SSL/TLS implementation during the initial messages, an adversary can obtain the user's account number and decrypt the PIN. The lack of randomness in this key again produces a vulnerability that can lead to user impersonation.

## 4.4 Access Control

A number of the applications that we analyzed used access control mechanisms that were poorly implemented or relied on incorrect or unverifiable assumptions that the user's device and its cellular communications channels are uncompromised. Multiple applications relied on SMS communications, but this channel is subject to a number of points of interception [56]. For example, another application on the device with the RECEIVE_SMS

[4] Although the Android offers a `SecureRandom` class for cryptographically-secure generation, it does not mention its necessity in the documentation.

permission could read the incoming SMS messages of the mobile money application. This functionality is outside the control of the mobile money application. Additionally, an attacker could have physical access to an unlocked phone, where messages can be inspected directly by a person. This channel does not, therefore, provide strong confidentiality or integrity guarantees.

**MoneyOnMobile.** The MoneyOnMobile app presents the most severe lack of access control we found among the apps we analyzed. The service uses two different PINs, the MPIN and TPIN, to authenticate the user for general functionality and transactions. However, we found that these PINs only prevent the user from moving between Android activities. In fact, the user's PINs are not required to execute any sensitive functionality via the backend APIs. All sensitive API calls (e.g., balance inquiry, mobile recharge, bill pay, etc.) except PIN changes can be executed with *only knowledge of the user's mobile phone number and two API calls.* MoneyOnMobile deploys no session identifiers, cookies, or other stateful tracking mechanisms during the app's execution; therefore, none of these are required to exploit the service.

The first required API call takes the mobile number as input and outputs various parameters of the account (e.g., Customer ID). These parameters identify the account as input in the subsequent API call. Due to the lack of any authentication on these sensitive functions, an adversary with no knowledge of the user's account can execute transactions on the user's behalf. Since the initial call provides information about a user account, this call allows an adversary to brute force phone numbers in order to find MoneyOnMobile users. This call also provides the remainder of the information needed to perform transactions on the account, severely compromising the security of the service.

**mPAY.** While the MoneyOnMobile servers do not require authentication before performing server tasks, we found the opposite is true with mPAY. The mPAY app accepts and performs unauthenticated commands from its server. The mPAY app uses a web/native app hybrid that allows the server to send commands to the app through the use of a URL parameter "method." These methods instruct the app to perform many actions, including starting the camera, opening the browser to an arbitrary URL, or starting an arbitrary app. If the control flow of the web application from the server side is secure, and the HTTP channel between client and server is free from injection or tampering, it is unlikely that these methods could be harmful. However, if an attacker can modify server code or redirect the URL, this functionality could be used to attack mobile users. Potential attacks include tricking users into downloading malware, providing information to a phishing website, or falling victim to a cross-site request forgery (CSRF) attack. As we discussed in the previous results, mPAY does not correctly validate the certificates used for its SSL/TLS sessions, and so these scenarios are unsettlingly plausible.

**GCash.** Although GCash implements authentication, it relies on easily-spoofable identity information to secure its accounts. During GCash's user registration process, the user selects a PIN for future authentication. The selected PIN is sent in plaintext over SMS along with the user's name and address. GCash then identifies the user with the phone number used to send the SMS message. This ties the user's account to their phone's SIM card. Unfortunately, SMS spoofing services are common, and these services provide the ability for an unskilled adversary to send messages appearing to be from an arbitrary number [27]. SIM cards can be damaged, lost, or stolen, and since the wallet balance is tied to this SIM, it may be difficult for a user to reclaim their funds.

Additionally, GCash requires the user to select a 4-digit PIN to register an account. As previously mentioned, this PIN is used to authenticate the user to the service. This allows only 10,000 possible combinations of PINs, which is quickly brute-forceable, though more intelligent guessing can be performed using data on the frequency of PIN selection [16]. We were not able to create an account with GCash to determine if the service locks accounts after a number of incorrect login attempts, which is a partial mitigation for this problem.

**Oxigen Wallet.** Like GCash, Oxigen Wallet also allows users to perform several sensitive actions via SMS. The most severe of these is requesting a new password. As a result, any attacker or application with access to a mobile phone's SMS subsystem can reset the password. That password can be used to login to the app or to send SMS messages to Oxigen Wallet for illicit transactions.

## 4.5 Information Leakage

Several of the analyzed applications exposed personally-identifying user information and/or data critical to the transactional integrity through various methods, including logging and preference storage.

### 4.5.1 Logging

The Android logging facility provides developers the ability to write messages to understand the state of their application at various points of its execution. These messages are written to the device's internal storage so they can be viewed at a future time. If the log messages were visible only to developers, this would not present the opportunity for a vulnerability. However, prior to Android 4.1, any application can declare the READ_LOGS permission and read the log files of any other application. That is, any arbitrary application (including malicious

ones) may read the logs. According to statistics from Google [32], 20.7% of devices run a version of Android that allows other apps to read logs.

**mPAY.** mPAY logs include user credentials, personal identifiers, and card numbers.

**GCash.** GCash writes the plaintext PIN using the verbose logging facility. The Android developer documentation states that verbose logging should not be compiled into production applications [33]. Although GCash has a specific `devLog` function that only writes this data when a debug flag is enabled, there are still statements without this check. Additionally, the session ID is also logged using the native Android logging facility without checking for a developer debug flag. An attacker with GCash log access can identify the user's PIN and the device ID, which could be used to impersonate the user.

**MoneyOnMobile.** These logs include server responses and account balances.

#### 4.5.2 Preference Storage

Android provides a separate mechanism for storing preferences. This system has the capability of writing the stored preferences to the device's local storage, where they can be recovered by inspecting the contents of the preferences file. Often, developers store preferences data in order to access it across application launches or from different sections of the code without needing to explicitly pass it. While the shared preferences are normally protected from the user and other apps, if the device is rooted (either by the user or a malicious application) the shared preferences file can be read.

**GCash.** GCash stores the user's PIN in this system. The application clears these preferences in several locations in the code (e.g., logout, expired sessions), however if the application terminates unexpectedly, these routines may not be called, leaving this sensitive information on the device.

**mPAY.** Similarly, mPAY stores the mobile phone number and customer ID in its preferences.

**mCoin.** Additionally, mCoin stores the user's name, birthday, and certain financial information such as the user's balance. We also found that mCoin exposes this data in transmission. Debugging code in the mCoin application is also configured to forward the user's mCoin shared preferences to the server with a debug report. As noted above, this may contain the user's personal information. This communication is performed over HTTP and sent in plaintext, providing no confidentiality for the user's data in transmission.

#### 4.5.3 Other Leakage

**Oxigen Wallet.** We discussed in Section 4.3 that requests from the Oxigen Wallet client are encrypted (insecurely) with either RSA or Blowfish. Oxigen Wallet also discloses mobile numbers of account holders. On sign up, Oxigen Wallet sends a `GetProfile` request to a server to determine if the mobile number requesting a new account is already associated with an email address. The client sends an email address, and the server sends a full mobile number back to the client. The application does appear to understand the security need for this data as only the last few digits of the mobile number are shown on the screen (the remaining digits are replaced by `Xs`). However, it appears that the full mobile number is provided in the network message. This means that if an attacker could somehow read the full message, he could learn the mobile number associated with the email address.

Unfortunately, the `GetProfile` request can be sent using the Blowfish encryption method previously described, meaning that an attacker could write his own code to poll the Oxigen Wallet servers to get mobile numbers associated with known email addresses. This enumeration could be used against a few targets or it may be done in bulk as a precursor to SMS spam, SMS phishing, or voice phishing. This bulk enumeration may also tax the Oxigen Wallet servers and degrade service for legitimate users. This attack would not be difficult for an attacker with even rudimentary programming ability.

### 4.6 Zuum

Zuum is a Brazilian mobile money application built by Mobile Financial Services, a partnership between Telefonica and MasterCard. While many of the other apps we analyzed were developed solely by cellular network providers or third-party development companies, MasterCard is an established company with experience building these types of applications.

This app is particularly notable because we did not find in Zuum the major vulnerabilities present in the other apps. In particular, the application uses SSL/TLS sessions with certificate validation enabled and includes a public key and performs standard cryptographic operations to protect transactions inside the session. Mallodroid detects Zuum's disabled certificate validation routines, but our manual analysis determines that these routines would not run. We discuss MasterCard's involvment in the Payment Card Industry standards, the app's terms of service, and the ramifications of compromise in Section 5.

## 4.7 Verification

We obtained accounts for MoneyOnMobile, Oxigen Wallet, and Airtel Money in India. For each app, we configured an Android emulator instance to forward its traffic through a man-in-the-middle proxy. In order to remain as passive as possible, we did not attempt to verify any transaction functionality (e.g., adding money to the account, sending or receiving money, paying bills, etc.). We were able to successfully verify every vulnerability that we identified for these apps.

# 5 Discussion

In this discussion section, we make observations about authentication practices and our SSL/TLS findings, regulations governing these apps, and whether smartphone applications are in fact safer than the legacy apps they replace.

*Why do these apps use weak authentication?* Numeric PINs were the authentication method of choice for the majority of the apps studied — only three apps allow use of a traditional password. This reliance on PINs is likely a holdover from earlier mobile money systems developed for feature phones. While such PINs are known to be weak against brute force attacks, they are chosen for SMS or USSD systems for two usability reasons. First, they are easily input on limited phone interfaces. Second, short numeric PINs remain usable for users who may have limited literacy (especially in Latin alphabets). Such users are far more common in developing countries, and prior research on secure passwords has assumed user literacy [54]. Creating a distinct strong password for the app may be confusing and limit user acceptability of new apps, despite the clear security benefits.

Beyond static PINs, Airtel Money and Oxigen Wallet (both based in India) use SMS-provided one-time passwords to authenticate users. While effective at preventing remote brute-force attacks, this step provides no defense against the other attacks we describe in the previous section.

*Why do these apps fail to validate certificates?* While this work and prior works have shown that many Android apps fail to properly validate SSL/TLS certificates [28], the high number of branchless banking apps that fail to validate certificates is still surprising, especially given the mission of these apps. Georgiev et al. found that many applications improperly validate certificates, yet identify the root cause as poorly designed APIs that make it easy to make a validation mistake [31]. One possible explanation is that certificate validation was disabled for a test environment which had no valid certificate. When the app was deployed, developers did not test for improper validation and did not remove the test code that disabled host name validation. Fahl et al. found this explanation to be common in developer interviews [29], and they also further explore other reasons for SSL/TLS vulnerabilities, including developer misunderstandings about the purpose of certificate validation.

In the absence of improved certificate management practices at the application layer, one possible defense is to enforce sane SSL/TLS configurations at the operating system layer. This capability is demonstrated by Fahl et al. for Android [29], while Bates et al. present a mechanism for Linux that simultaneously facilitates the use of SSL trust enhancements [15]. In the event that the system trusts compromised root certificates, a solution like DVCert [23] could be used to protect against man in the middle attacks.

*Are legacy systems more secure?* In Section 7, we noted that prior work had found that legacy systems are fundamentally insecure as they rely principally on insecure GSM bearer channels. Those systems rely on bearer channel security because of the practical difficulties of developing and deploying secure applications to a plethora of feature phone platforms with widely varying designs and computational capabilities. In contrast, we look at apps developed for relatively homogenous, well-resourced smartphones. One would expect that the advanced capabilities available on the Android platform would increase the security of branchless banking apps. However, given the vulnerabilities we disclose, the branchless banking apps we studied for Android put users at a *greater* risk than legacy systems. Attacking cellular network protocols, while shown to be practical [56], still has a significant barrier to entry in terms of equipment and expertise. In contrast, the attacks we disclose in this paper require only a laptop, common attack tools, and some basic security experience to discover and exploit. Effectively, these attacks are easier to exploit than the previously disclosed attacks against SMS and USSD interfaces.

*Does regulation help?* In the United States, the PCI Security Standards Council releases a Data Security Standard (PCI DSS) [48], which govern the security requirements for entities that handle cardholder data (e.g., card numbers and expiration dates). The council is a consortium of card issuers including Visa, MasterCard, and others that cooperatively develop this standard. Merchants that accept credit card payments from these issuers are generally required to adhere to the PCI DSS and are subject to auditing.

The DSS document includes requirements, testing procedures, and guidance for securing devices and net-

works that handle cardholder data. These are not, however, specific enough to include detailed implementation instructions. The effectiveness of these standards is not our main focus; we note that the PCI DSS can be used as a checklist-style document for ensuring well-rounded security implementations.

In 2008, the Reserve Bank of India (RBI) issued guidelines for mobile payment systems [13]. (By their definition, the apps we study would be included in these guidelines). In 12 short pages, they touch on aspects as broad as currencies allowed, KYC/AML policies, interbank settlement policies, corporate governance approval, legal jurisdiction, consumer protection, and technology and security standards for a myriad of delivery channels. The security standards give implementers wide leeway to use their best judgement about specific security practices. MoneyOnMobile, which had the most severe security issues among all of the apps we manually analyzed, prominently displays its RBI authorization on its web site.

Some prescriptions stand out from the rest: an objective to have "digital certificate based inquiry/transaction capabilities," a recommendation to have a mobile PIN that is encrypted on the wire and never stored in cleartext, and use of the mobile phone number as the chief identifier. These recommendations may be responsible for certain design decisions of Airtel Money and Oxigen Wallet (both based in India). For example, the digital certificate recommendation may have driven Oxigen Wallet developers to develop their (very flawed) public key encryption architecture. These recommendations also explain why Airtel Money elected to further encrypt the PIN (and only the PIN) in messages that are encapsulated by TLS. Further, the lack of guidance on what "strong encryption" entails may be partially responsible for the security failures of Airtel Money and Oxigen Wallet. Finally, we note that we believe that Airtel Money, while still vulnerable, was within the letter of the recommendations.

To our knowledge, other mobile money systems studied in this paper are not subject to such industry or government regulation. While a high-quality, auditable industry standard may lead to improved branchless banking security, it is not clear that guidelines like RBI's currently make much of a difference.

## 6 Terms of Service & Consumer Liability

After uncovering technical vulnerabilities for branchless banking, we investigated their potential implications for fraud liability. In the United States, the consumer is not held liable for fraudulent transactions beyond a small amount. This model recognizes that users are vulnerable to fraud that they are powerless to prevent, combat, or detect prior to incurring losses.

To determine the model used for the branchless banking apps we studied, we surveyed the Terms of Service (ToS) for each of the seven analyzed apps analyzed. The Airtel Money [1], GCash [3], mCoin [5], Oxigen Wallet [9], MoneyOnMobile [7], and Zuum [12] terms all hold the customer solely responsible for most forms of fraudulent activity. Each of these services hold the customer responsible for the safety and security of their password. GCash, mCoin, and Oxigen Wallet also hold the customer responsible for protecting their SIM (i.e., mobile phone). GCash provides a complaint system, provided that the customer notifies GCash in writing within 15 days of the disputed transaction. However, they also make it clear that erroneous transactions are not grounds for dispute. mPAY's terms [8] are less clear on the subject of liability; they provide a dispute resolution system, but do not detail the circumstances for which the customer is responsible. Across the body of these terms of service, it is overwhelmingly clear that the customer is responsible for all transactions conducted with their PIN/password on their mobile device.

The presumption of customer fault for transactions is at odds with the findings of this work. The basis for these arguments appear to be that, if a customer protects their PIN and protects their physical device, there is no way for a third party to initiate a fraudulent transaction. We have demonstrated that this is not the case. Passwords can be easily recovered by an attacker. Six of the seven apps we manually analyzed transmits authentication data over insecure connections, allowing them to be recovered in transit. Additionally, with only brief access to a customer's phone, an attacker could read GCash PINs out of the phone logs or trigger the Oxigen Wallet password recovery mechanism. Even when the mobile device and SIM card are fully under customer control, unauthorized transactions can still occur, due to the pervasive vulnerabilities found in these six apps. By launching a man-in-the-middle attack, an adversary would be able to tamper with transactions while in transit, misleading the provider into believing that a fraudulent transaction originated from a legitimate user. *These attacks are all highly plausible.* Exploits of the identified vulnerabilities are not probabilistic, but would be 100% effective. With only minimal technical capability, an adversary could launch these attacks given the ability to control a local wireless access point. This litany of vulnerabilities comes only from an analysis of client-side code. Table 2 hints that there may be further server side configuration issues, to say nothing of the security of custom server software, system software, or the operating systems used.

Similar to past findings for the "Chip & Pin" credit card system [40], it is possible that these apps are already being exploited in the wild, leaving consumers with no

recourse to dispute fraudulent transactions. Based on the discovery of rampant vulnerabilities in these applications, we feel that the liability model for branchless banking applications must be revisited. Providers must not marry such vulnerable systems with a liability model that refuses to take responsibility for the technical flaws, and these realities could prevent sustained growth of branchless banking systems due to the high likelihood of fraud.

## 7 Related Work

Banking has been a motivation for computer security since the origins of the field. The original Data Encryption Standard was designed to meet the needs of banking and commerce, and Anderson's classic paper "Why Cryptosystems Fail" looked specifically at banking security [14]. Accordingly, mobile money systems have been scrutinized by computer security practitioners. Current research on mobile money systems to-date has focused on the challenges of authentication, channel security, and transaction verification in legacy systems designed for feature phones. Some prior work has provided threat modeling and discussion of broader system-wide security issues. To our knowledge, we are the first to examine the security of smartphone applications used by mobile money systems.

Mobile money systems rely on the network to provide identity services; in essence, identity is the telephone number (MS-ISDN) of the subscriber. To address physical access granting attackers access to accounts, researchers have investigated the use of a small one-time pads as authenticators in place of PINs. Panjwani et al. [47] present a new scheme that avoids vulnerabilities with using one-time passwords with PINs and SMS. Sharma et al. propose using scratch-off one-time authenticators for access with supplemental recorded voice confirmations [53]. These schemes add complexity to the system while only masking the PIN from an adversary who can see a message. These schemes do not provide any guarantees against an adversary who can modify messages or who recovers a message and a pad.

SMS-based systems, in particular, are vulnerable to eavesdropping or message tampering [42], and so have seen several projects to bring additional cryptographic mechanisms to mobile money systems [20, 41, 22]. Systems that use USSD, rather than SMS, as their bearer channel can also use code executing on the SIM card to cryptographically protect messages. However, it is unknown how these protocols are implemented or what guarantees they provide [45].

Finally, several authors have written papers investigating the holistic security of mobile money systems designed exclusively for "dumbphones." Paik et al. [45] note concerns about reliance on GSM traffic channel cryptographic guarantees, including the ability to intercept, replay, and spoof the source of SMS messages. Panjwani fulfills the goals laid out by Paik et al. by providing a brief threat model and a design to protect against the threats they identify [46]. While those papers focus on technical analysis, de Almeida [38] and Harris et al. [35] note the policy implications of the insecurity of mobile money.

While focused strictly on mobile money platforms, this paper also contributes to the literature of Android application security measurement. The pioneering work in this space was TaintDroid [25, 25], a dynamic analysis system that detected private information leakages. Shortly after, Felt et al. found that one-third of apps studied held privileges they did not need [30], while Chin et al. found that 60% of apps manually examined were vulnerable to attacks involving Android Intents [19]. More recently, Fahl et al. [28] and Egele et al. [24] use automated static analysis investigated cryptographic API use in Android, finding respectively that 8% of apps studied were vulnerable to man-in-the-middle attacks and that 88% of apps make some mistake with regards to cryptographic libraries [24]. Our work confirms these results apply to mobile money applications. This project is most similar to the work of Enck et al. [26], who automatically and manually analyzed 1,100 applications for a broad range of security concerns.

However, prior work does not investigate the security guarantees and the severe consequences of smart phone application compromise in branchless banking systems. Our work specifically investigates this open area of research and provides the world's first detailed security analysis of mobile money apps. In doing so, we demonstrate the risk to users who rely on these systems for financial security.

## 8 Conclusions

Branchless banking applications have and continue to hold the promise to improve the standard of living for many in the developing world. By enabling access to a cashless payment infrastructure, these systems allow residents of such countries to reap the benefits afforded to modern economies and decrease the physical security risks associated with cash transactions. However, the security of the applications providing these services has not previously been vetted in a comprehensive or public fashion. In this paper, we perform precisely such an analysis on seven branchless banking applications, balancing both popularity with geographic representation. Our analysis targets the registration, login, and transaction portions of the representative applications, and codifies discovered vulnerabilities using the CWE classification system. We find significant vulnerabilities in six

of the seven applications, which prevent both users and providers from reasoning about the integrity of transactions. We then pair these technical findings with the discovery of fraud liability models that explicitly hold the end user culpable for all fraud. Given the systemic problems we identify, we argue that dramatic improvements to the security of branchless banking applications are imperative to protect the mission of these systems.

## Acknowledgments

The authors are grateful to Saili Sahasrabudde for her assistance with this work. We would also like to thank the members of the SENSEI Center at the University of Florida for their help in preparing this work, as well as our anonymous reviewers for their helpful comments.

This work was supported in part by the US National Science Foundation under grant numbers CNS-1526718, CNS-1540217, and CNS-1464087. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] Airtel Money: Terms and Conditions of Usage. `http://www.airtel.in/personal/money/terms-of-use`.

[2] android-apktool: A Tool for Reverse Engineering Android APK Files. `https://code.google.com/p/android-apktool/`.

[3] GCash Terms and Conditions. `http://www.globe.com.ph/gcash-terms-and-conditions`.

[4] JEB Decompiler. `http://www.android-decompiler.com/`.

[5] mCoin: Terms and Conditions. `http://www.mcoin.co.id/syarat-dan-ketentuan`.

[6] MMU Deployment Tracker. `http://www.gsma.com/mobilefordevelopment/programmes/mobile-money-for-the-unbanked/insights/tracker`.

[7] Money on Mobile Sign-Up: Terms and Conditions. `http://www.money-on-mobile.com`.

[8] mPAY: Terms and Conditions. `http://www.ais.co.th/mpay/en/about-term-condition.aspx`.

[9] Oxigen Wallet: Terms and Conditions. `https://www.oxigenwallet.com/terms-conditions`.

[10] smali: An assembler/disassembler for Android's dex format. `https://code.google.com/p/smali/`.

[11] The Legion of the Bouncy Castle. `https://www.bouncycastle.org/`.

[12] Zuum: Termos e Condições. `http://www.zuum.com.br/institucional/termos`.

[13] Mobile Payment in India — Operative Guidelines for Banks. Technical report, Reserve Bank of India, 2008.

[14] R. Anderson. Why Cryptosystems Fail. In *Proc. of the 1st ACM Conf. on Comp. and Comm. Security*, pages 215–227. ACM Press, 1993.

[15] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, A. Alkhelaifi, and K. Butler. Securing SSL Certificate Verification through Dynamic Linking. In *Proc. of the 21st ACM Conf. on Comp. and Comm. Security (CCS'14)*, Scottsdale, AZ, USA, Nov. 2014.

[16] N. Berry. PIN analysis. `http://www.datagenetics.com/blog/september32012/`, Sept. 2012.

[17] Bill & Melinda Gates Foundation. Financial Services for the Poor: Strategy Overview. `http://www.gatesfoundation.org/What-We-Do/Global-Development/Financial-Services-for-the-Poor`.

[18] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proc. 36th Intl. Conf. Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[19] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-application Communication in Android. In *Proc. 9th Intl. Conf. Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[20] M. K. Chong. *Usable Authentication for Mobile Banking*. PhD thesis, Univ. of Cape Town, Jan. 2009.

[21] P. Chuhan-Pole and M. Angwafo. Mobile Payments Go Viral: M-PESA in Kenya. In *Yes, Africa Can: Success Stories from a Dynamic Continent*. World Bank Publications, June 2011.

[22] S. Cobourne, K. Mayes, and K. Markantonakis. Using the Smart Card Web Server in Secure Branchless Banking. In *Network and System Security*, number 7873 in Lecture Notes in Computer Science, pages 250–263. Springer Berlin Heidelberg, Jan. 2013.

[23] I. Dacosta, M. Ahamad, and P. Traynor. Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties. In *Proceedings of the European Symposium on Research in Computer Security*, pages 199–216. Springer, 2012.

[24] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proc. 20th ACM Conf. Comp. and Comm. Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[25] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[26] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. 20th USENIX Security Sym.*, San Francisco, CA, USA, 2011.

[27] W. Enck, P. Traynor, P. McDaniel, and T. La Porta. Exploiting open functionality in SMS-capable cellular networks. In *Proc. of the 12th ACM conference on Comp. and communications security*, pages 393–404. ACM, 2005.

[28] S. Fahl, M. Harbach, T. Muders, L. Baumgartner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proc. 2012 ACM Conf. Comp. and Comm. Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[29] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking SSL Development in an Appified World. In *Proc. 20th ACM Conf. Comp. and Comm. Security*, CCS '13, pages 49–60, New York, NY, USA, 2013. ACM.

[30] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proc. 18th ACM Conf. Comp. and Comm. Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[31] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proc. 2012 ACM Conf. Comp. and Comm. Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[32] Google. Dashboards — Android Developers. `https://developer.android.com/about/dashboards/index.html`.

[33] Google. Log — Android Developers. `https://developer.android.com/reference/android/util/Log.html`.

[34] Google. Random — Android Developers. `https://developer.android.com/reference/java/util/Random.html`.

[35] A. Harris, S. Goodman, and P. Traynor. Privacy and Security Concerns Assocaited with Mobile Money Applications in Africa. *Washington Journal of Law, Technology & Arts*, 8(3), 2013.

[36] V. Highfield. More than 60 Per Cent of Kenyan GDP Came From Mobile Money in June 2012, a New Survey Shows. `http://www.totalpayments.org/2013/03/01/60-cent-kenyan-gdp-mobile-money-june-2012-survey-shows/`, 2012.

[37] J. Kamana. M-PESA: How Kenya Took the Lead in Mobile Money. `http://www.mobiletransaction.org/m-pesa-kenya-the-lead-in-mobile-money/`, Apr. 2014.

[38] G. Martins de Almeida. M-Payments in Brazil: Notes on How a Countrys Background May Determine Timing and Design of a Regulatory Model. *Washington Journal of Law, Technology & Arts*, 8(3), 2013.

[39] C. Mims. 31% of Kenya's GDP is Spent Through Mobile Phones. `http://qz.com/57504/31-of-kenyas-gdp-is-spent-through-mobile-phones/`, Feb. 2013.

[40] S. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and PIN is Broken. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 433–446, May 2010.

[41] B. W. Nyamtiga, A. Sam, and L. S. Laizer. Enhanced security model for mobile banking systems in Tanzania. *Intl. Jour. Tech. Enhancements and Emerging Engineering Research*, 1(4):4–20, 2013.

[42] B. W. Nyamtiga and L. S. Sam, Anael Laizer. Security perspectives for USSD versus SMS in conducting mobile transactions: A case study of Tanzania. *Intl. Jour. Tech. Enhancements and Emerging Engineering Research*, 1(3):38–43, 2013.

[43] J. Ong. Android Achieved 85% Smartphone Market Share in Q2. `http://thenextweb.com/google/2014/07/31/android-reached-record-85-smartphone-market-share-q2-2014-report/`, July 2014.

[44] Oracle. Random - Java Platform SE 7. `https://docs.oracle.com/javase/7/docs/api/java/util/Random.html`.

[45] M. Paik. Stragglers of the Herd Get Eaten: Security Concerns for GSM Mobile Banking Applications. In *Proc. 11th Workshop on Mobile Comp. Syst. and Appl.*, HotMobile '10, pages 54–59, New York, NY, USA, 2010. ACM.

[46] S. Panjwani. Towards End-to-End Security in Branchless Banking. In *Proc. 12th Workshop on Mobile Comp. Syst. and Appl.*, HotMobile '11, pages 28–33, New York, NY, USA, 2011. ACM.

[47] S. Panjwani and E. Cutrell. Usably Secure, Low-Cost Authentication for Mobile Banking. In *Proc. 6th Symp. Usable Privacy and Security*, SOUPS '10, pages 4:1–4:12, New York, NY, USA, 2010. ACM.

[48] PCI Security Standards Council, LLC. Data Security Standard — Requirements and Security Assessment Procedures. `https://www.pcisecuritystandards.org/documents/PCI_DSS_v3.pdf`.

[49] C. Penicaud and A. Katakam. Mobile Financial Services for the Unbanked: State of the Industry 2013. Technical report, GSMA, Feb. 2014.

[50] Qualys. SSL Server Test. `https://www.ssllabs.com/ssltest/`.

[51] Reserve Bank of India. Master Circular - KYC norms, AML standards, CFT, Obligation of banks under PMLA, 2002. `http://rbidocs.rbi.org.in/rdocs/notification/PDFs/94CF010713FL.pdf`, 2013.

[52] Safaricom. Relax, you have got M-PESA. `http://www.safaricom.co.ke/personal/m-pesa/m-pesa-services-tariffs/relax-you-have-got-m-pesa`.

[53] A. Sharma, L. Subramanian, and D. Shasha. Secure Branchless Banking. In *3rd ACM Workshop on Networked Syst. for Developing Regions*, Big Sky, Montana, Oct. 2009.

[54] R. Shay, S. Komanduri, A. L. Durity, P. S. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor. Can Long Passwords Be Secure and Usable? In *Proc. Conf. on Human Factors in Comp. Syst.*, CHI '14, pages 2927–2936, New York, NY, USA, 2014. ACM.

[55] The MITRE Corporation. CWE - Common Weakness Enumeration. `http://cwe.mitre.org/`.

[56] P. Traynor, P. McDaniel, and T. La Porta. *Security for Telecommunications Networks*. Springer, 2008.

# Appendix

| Package Name | Country | Downloads | Mallodroid Alert |
|---|---|---|---|
| bo.com.tigo.tigoapp | Bolivia | 1000-5000 | |
| br.com.mobicare.minhaoi | Brazil | 500000-1000000 | ✗ |
| com.cellulant.wallet | Nigeria | 100-500 | ✗ |
| com.directoriotigo.hwm | Honduras | 10000-50000 | |
| com.econet.ecocash | Zimbabwe | 10000-50000 | |
| com.ezuza.mobile.agent | Mexico | 10-50 | |
| com.f1soft.esewa | Nepal | 50000-100000 | |
| com.fetswallet_App | Nigeria | 100-500 | |
| com.globe.gcash.android | Philippines | 10000-50000 | ✗ |
| com.indosatapps.dompetku | Indonesia | 5000-10000 | ✗ |
| com.japps.firstmonie | Nigeria | 50000-100000 | |
| com.m4u.vivozuum | Brazil | 10000-50000 | ✗ |
| com.mcoin.android | Indonesia | 1000-5000 | ✗ |
| com.mdinar | Tunisia | 500-1000 | ✗ |
| com.mfino.fortismobile | Nigeria | 100-500 | ✗ |
| com.mibilleteramovil | Argentina | 500-1000 | |
| com.mobilis.teasy.production | Nigeria | 100-500 | |
| com.mom.app | India | 10000-50000 | |
| com.moremagic.myanmarmobilemoney | Myanmar | 191 | |
| com.mservice.momotransfer | Vietnam | 100000-500000 | ✗ |
| com.myairtelapp | India | 1000000-5000000 | ✗ |
| com.oxigen.oxigenwallet | India | 100000-500000 | ✗ |
| com.pagatech.customer.android | Nigeria | 1000-5000 | |
| com.palomar.mpay | Thailand | 100000-500000 | ✗ |
| com.paycom.app | Nigeria | 10000-50000 | ✗ |
| com.pocketmoni.ui | Nigeria | 5000-10000 | |
| com.ptdam.emoney | Indonesia | 100000-500000 | Market Restriction |
| com.qulix.mozido.jccul.android | Jamaica | 1000-5000 | ✗ |
| com.sbg.mobile.phone | South Africa | 100000-500000 | N/A |
| com.simba | Lebanon | 1000-5000 | ✗ |
| com.SingTel.mWallet | Singapore | 100000-500000 | ✗ |
| com.suvidhaa.android | India | 10000-50000 | Market Restriction |
| com.tpago.movil | Dominican Republic | 5000-10000 | ✗ |
| com.useboom.android | Mexico | 5000-10000 | ✗ |
| com.vanso.gtbankapp | Nigeria | 100000-500000 | |
| com.wizzitint.banking | South Africa | 100-500 | ✗ |
| com.zenithBank.eazymoney | Nigeria | 50000-100000 | ✗ |
| mg.telma.mvola.app | Madagascar | 1000-5000 | N/A |
| net.omobio.dialogsc | Sri Lanka | 50000-100000 | ✗ |
| org.readycash.android | Nigeria | 1000-5000 | |
| qa.ooredoo.omm | Qatar | 5000-10000 | |
| sv.tigo.mfsapp | El Salvador | 10000-50000 | ✗ |
| Tag.Andro | Côte d'Ivoire | 500-1000 | |
| th.co.truemoney.wallet | Thailand | 100000-500000 | ✗ |
| tz.tigo.mfsapp | Tanzania | 50000-100000 | ✗ |
| uy.com.antel.bits | Uruguay | 10000-50000 | |
| com.vtn.vtnmobilepro | Nigeria | Unknown | |
| za.co.fnb.connect.itt | South Africa | 500000-1000000 | |

Table 3: We found 48 mobile money Android applications across 28 countries. Highlighted rows represent those applications manually analyzed in this paper. We were unable to obtain two apps due to Android market restrictions. Mallodroid was unable to analyze the apps marked N/A.