Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications

BRADLEY REAVES, JASMINE BOWERS, NOLEN SCAIFE, University of Florida
ADAM BATES, University of Illinois at Urbana-Champaign
ARNAV BHARTIYA, PATRICK TRAYNOR, and KEVIN R. B. BUTLER, University of Florida

Mobile money, also known as branchless banking, leverages ubiquitous cellular networks to bring much-needed financial services to the unbanked in the developing world. These services are often deployed as smartphone apps, and although marketed as secure, these applications are often not regulated as strictly as traditional banks, leaving doubt about the truth 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 from 2015. We then perform a comprehensive manual teardown of the registration, login, and transaction procedures of a diverse 15% of these apps. We uncover pervasive vulnerabilities spanning botched certification validation, do-it-yourself cryptography, and other forms of information leakage that allow an attacker to impersonate legitimate users, modify transactions, and steal financial records. These findings show that the majority of these apps fail to provide the protections needed by financial services. In an expanded reevaluation one year later, we find that these systems have only marginally improved their security. Additionally, we document our experiences working in this sector for future researchers and provide recommendations to improve the security of this critical ecosystem. 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 raise the standard of living and have revolutionized the way money is used in developing economies. Over 30% of the GDP in many nations can now be attributed to branchless banking applications [Mims 2013], many of which now perform more transactions per month than traditional payment processors, including PayPal [Highfield 2012].

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 longitudinal and comprehensive security analysis of branchless banking applications, providing insight into a year of mobile money app development. 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 in 2015. We then methodically select seven Android-based branchless banking applications from Brazil, India, Indonesia, Thailand, and the Philippines 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. While some of the individual vulnerabilities we study have been examined in previous work [Fahl et al. 2012; Egele et al. 2013], this work takes a holistic measurement of a critical industry.

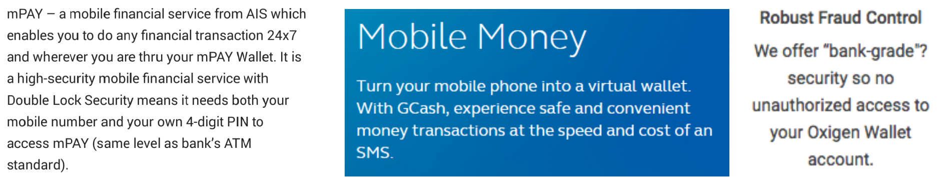
— **Identification of Systemic Vulnerabilities:** Our analysis discovers pervasive weaknesses and shows that six of the seven applications broadly fail to preserve the integrity of their transactions. We then compare our results to those provided through automated analysis and show that current tools 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.
In an expanded follow up study conducted one year after the initial analysis, we reevaluate the apps analyzed and find that a significant number of vulnerabilities still remain in the applications.

— **Analysis of Liability, Public Response, and Recommendations:** 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. Additionally, we provide insights for future researchers by discussing the vendor and public response that resulted from our initial analysis. We conclude with recommendations for vendors and platforms to help improve the security of mobile money.

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 of broader implications of our findings; Section 6 offers an analysis of the Terms of Service and the assignment of liability; Section 7 provides an account of the vendor and public response from our initial study; Section 8 discusses our follow up analysis of application security one year after the results from Section 4; Section 9 presents lessons learned and recommendations for mobile money operators; Section 10 discusses relevant related work; and Section 11 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 [Bill & Melinda Gates Foundation 2016]. 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 setbacks, conduct commerce remotely, or protect money against loss or theft. Accordingly, providing


mPAY – a mobile financial service from AIS which enables you to do any financial transaction 24x7 and wherever you are thru your mPAY Wallet. It is a high-security mobile financial service with Double Lock Security means it needs both your mobile number and your own 4-digit PIN to access mPAY (same level as bank's ATM standard).

Mobile Money

Turn your mobile phone into a virtual wallet. With GCash, experience safe and convenient money transactions at the speed and cost of an SMS.

Robust Fraud Control

We offer "bank-grade"? security so no unauthorized access to your Oxigen Wallet account.

(a) mPAY (b) GCash (c) Oxigen Wallet

Fig. 1: Mobile money apps are heavily marketed as being safe to use. These screenshots from providers' marketing materials show the extent of these claims. Note that the question mark in Oxigen Wallet's screenshot was present on the original website.

banking through mobile phone networks offers the promise of dramatically improving the lives of the world's poor.

The M-Pesa system in Kenya [Chuhan-Pole and Angwafo 2011] 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, uneconomical due to minimum balances, or simply non-existent. M-Pesa has not yet released a mobile app, but it 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 [Kamana 2014]. In Kenya and at least 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 [Penicaud and Katakam 2014] 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 of these promises. This promise of financial security is even reflected in the M-Pesa advertising slogan "Relax, you have got M-Pesa." [Safaricom 2014]. However, the veracity of these claims is unknown.

2.1. Comparison to Other Services

Mobile money is closely related to other technologies.

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, and Square Cash. These systems act 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 (e.g., Oxigen Wallet) and mobile payment systems (e.g., 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 [Reserve Bank of India 2013]. 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 [Traynor et al. 2008].

3. APP SELECTION AND ANALYSIS

In this section, we discuss how apps were chosen for analysis and how our initial analysis was conducted in 2015. As part of our longitudinal study, we reevaluated these apps in 2016. Section 8 discusses app selection and analysis conducted during the reevaluation in 2016.

3.1. Mallodroid Analysis

Using data from the GSMA Mobile Money Tracker [GSMA 2014], we identified a total of 48 Android mobile money apps across 28 countries. We first ran an automated analysis on all 48 of these apps using Mallodroid [Fahl et al. 2012], a static analysis tool for detecting TLS vulnerabilities, in order to establish a baseline. Table V 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* TLS. It similarly cannot detect ecosystem vulnerabilities, including the use of deprecated, vulnerable, or incorrect TLS configurations on remote servers. Finally, the absence of 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; only 41% of those apps were vulnerable to TLS man-in-the-middle attacks. Thus it is imperative to further verify the findings of this tool to remove false positives and false negatives.

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 number of 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 2 shows the geographic diversity of all the mobile money apps we analyze.

[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.

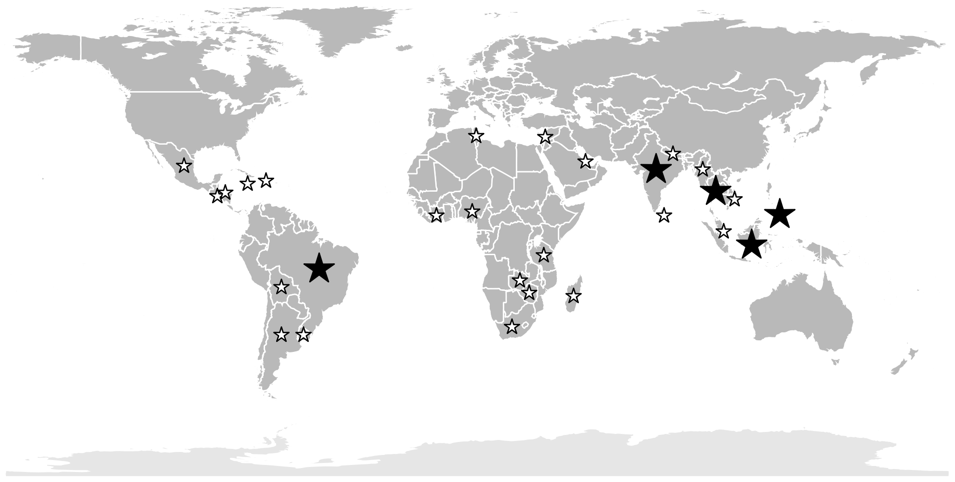


Fig. 2: The mobile money applications we analyzed were developed for a diverse range of countries. In 2015, we performed an initial analysis on applications from 28 countries representing up to approximately 1.2 million users based on market download counts. From this, we selected 7 applications to fully analyze from 5 countries. In 2016, we reanalyzed the selected apps to determine if the originally discovered vulnerabilities still remained. Each black star represents these countries, and the white stars represent the remainder of the mobile money systems.

We focus on Android applications in this paper because Android has the largest market share worldwide [Ong 2014], 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 most 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 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 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 app's code and manifest and testing the quality of any 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. If we had an account for the app, we executed it 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 [Enck et al. 2011], 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. The first step of the analysis was to obtain the application manifest using `apktool` [Apktool 2016]. We then used a simple script to generate a report identifying each app component (i.e., activities, services, content providers, and 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` [Smali 2014] 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 [Bouncy Castle 2014]), well-known advertising libraries (as identified by Chen et al. [Chen et al. 2014]), 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 (e.g., analytics libraries, user interface code, HTTP libraries, etc.).

While analyzing the app's code can provide 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, servers can offer vulnerable versions of TLS, weak signature algorithms, and/or expired or invalid certificates. Thus, the final step of the analysis was to check each application's TLS endpoints using the Qualys SSL Server Test [Qualys 2014].[2] This test provides a comprehensive view of the configuration and capabilities of a server's TLS implementation.

Phase 2: Reverse Engineering

In order to complete our holistic view of both the application protocols and the client/server TLS negotiation, we reverse engineered each app in the second phase. We used the commercial interactive JEB Decompiler [JEB Decompiler 2014] 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 the 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`

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

[3]We found several vulnerabilities in 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 Auth.			✗			✗	
Access Control								
CWE-88	Argument Injection or Modification		✗					
CWE-302	Auth. 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			✗				
Info. Leakage								
CWE-200	Info. Exposure			✗			✗	✗
CWE-532	Info. Exposure Through Log Files		✗		✗		✗	
CWE-312	Cleartext Storage of Sensitive Info.		✗		✗			✗
CWE-319	Cleartext Transmission of Sensitive Info.			✗	✗		✗	

Table I: Weaknesses in Mobile Money Applications, indexed to corresponding Common Weakness Enumeration (CWE) categories. The CWE database is a comprehensive taxonomy of software vulnerabilities developed by MITRE [The MITRE Corporation 2014] and provide a common language for software errors.

user interface code, we also analyze other components that 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, and 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), or practices that discourage good security hygiene (e.g., permitting insecure passwords). We discuss our specific findings in Section 4.

3.3.1. Vulnerability Disclosure. All vulnerabilities in the following section were responsibly disclosed to the services before public disclosure. Section 7 discusses this disclosure and the vendor response in detail.

4. RESULTS

This section details the results of analyzing the mobile money applications. Overall, we find 28 significant vulnerabilities across seven applications. Table I 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. We assigned a unique identifier to each vulnerability found during our analysis (e.g., V1: PIN logged in plaintext). Table IV in Section 8 contains the complete list of vulnerabilities and corresponding identifiers.

4.1. Automated Analysis

Our results for TLS vulnerabilities should mirror the output of a 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.

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 II: 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.

Mallodroid produces a false positive when it detects a 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 checks for development modes.

Conversely, in the case of MoneyOnMobile, Mallodroid produces a false negative. MoneyOnMobile contains no TLS vulnerability because it does not employ 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 TLS in an application. It produces an alert for the most secure app we analyzed and did not produce an alert 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 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. TLS

As we discussed above, problems with 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 exceptions on invalid certificates. This insecure practice was previously identified by Georgiev et al. [2012] and is specifically targeted by Mallodroid.

Analyzing only the app does not provide complete visibility to the overall security state of a TLS session. Server misconfiguration can introduce additional vulnerabilities, even when the client application uses correctly implemented TLS. To account for this, we also ran the Qualys SSL Server Test [Qualys 2014] on each of the HTTPS endpoints we discovered while analyzing the apps. This service tests a number of prop-

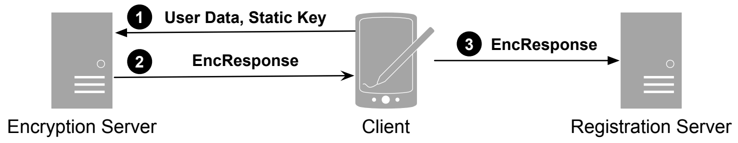


Fig. 3: The user registration flow of MoneyOnMobile. All communication is over HTTP.

erties of each server to identify configuration and implementation errors and provide a "grade" for the configuration. These results are presented in Table II. Three of the endpoints we tested received failing scores due to insecure implementations of 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 (V5) 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) (V6). *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 TLS sessions, a man-in-the-middle adversary is free to manipulate the inside messages.

4.3. Non-Standard Cryptography

Despite the pervasive insecure implementations of 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 TLS. All API calls from the app use HTTP (V8, V18). 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 3.

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 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 (V10). Instead, only 17 bits of the key are random. The remaining bits are filled by the mobile phone number, the date, and padding with `0`s. The random bits are generated by the `Random` [Google 2014c] random number generator. The standard Java documentation [Oracle 2014] *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 (V11). 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 (V12). 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 (V13). 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 communications with the remote server (V4). 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 (V3). 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, the IMEI is frequently abused by mobile apps for persistent tracking of users [Enck et al. 2014], 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

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

app does not validate certificates (V5), 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 (V16) that concatenates the device's phone number and account number in the following format:

$$Key_{enc} = \texttt{j7zgy1yv} \parallel phone\# \parallel account\# \tag{1}$$

The prefixed text in the key is immutable and included with the application. Due to the weak 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 [Traynor et al. 2008]. For example, another application on the device with the RECEIVE_SMS 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 (V9).* 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 (V17). 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 (V5) used for its 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 (V2) 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 [Enck et al. 2005]. 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 [Berry 2012]. 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 (V15). 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 transaction 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 [Google 2014a], 20.7% of devices run a version of Android that allows other apps to read logs.

mPAY. mPAY logs include user credentials, identifiers, and card numbers (V1).

GCash. GCash writes the plaintext PIN using the verbose logging facility (V1). The Android developer documentation states that verbose logging should not be compiled into production applications [Google 2014b]. 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 (V1).

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 (V0). 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. mPay stores mobile phone numbers and customer IDs in its preferences (V0).
mCoin. Additionally, mCoin stores the user's name, birthday, and certain financial information such as the user's balance (V0). 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 (V7). 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 (V14). 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 `X`s). 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 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 involvement 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 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 [Shay et al. 2014]. 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 TLS certificates [Fahl et al. 2012], 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. [2012] 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. 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. [2013] found this explanation to be common in developer interviews, and they also explore other reasons for TLS vulnerabilities, including developer misunderstandings about certificate validation.

In the absence of improved certificate management practices at the application layer, one possible defense is to enforce sane TLS configurations at the operating system layer. This capability is demonstrated by Fahl et al. [2013] for Android, while Bates et al. [2014] present a mechanism for Linux that simultaneously facilitates the use of SSL trust enhancements. In the event that the system trusts compromised root certificates, a solution like DVCert [Dacosta et al. 2012] could be used to protect against man in the middle attacks.

Are legacy systems more secure? In Section 10, 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 [Traynor et al. 2008], 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 released a Data Security Standard (PCI DSS) [PCI DSS 2016], 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 networks 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 [Reserve Bank of India 2008]. (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, inter-bank 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 judgment 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. The Airtel Money [Airtel TOS 2015], GCash [GCash TOS 2015], mCoin [mCoin TOS 2015], Oxigen Wallet [Oxigen TOS 2015], MoneyOnMobile [Money on Mobile TOS 2015], and Zuum [Zuum TOS 2015] 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 [mPay TOS 2015] 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. This remains true in August 2016.

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 III 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 [Murdoch et al. 2010], 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. Many of the vulnerabilities and attacks in this paper have been found in applications outside of the mobile money industry, and mobile money applications face equivalent *technical* threat models to banking applications in the developing world. However, because these models place all of the liability of compromise on the end user, users of these systems are exposed to risk of much greater losses. 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. VENDOR AND PUBLIC RESPONSE

We published the first version of this work at the USENIX Security Symposium in August 2015 [Reaves et al. 2015]. Two months prior to the public release of the paper, we contacted each of the vendors of the six manually analyzed applications. We provided each vendor with a detailed description of the vulnerabilities and practical fixes to the problems we discovered. We shared these detailed vulnerability reports only with the vendors, and have not made their specific contents (e.g., method names, recovered source files, etc.) public.

The act of informing the vendors itself proved difficult. Only one of the vendors had a publicly-listed security contact. For the majority of the remaining applications, we were forced to submit these reports to multiple generic contact email addresses listed on either their home pages or within the app market. Our attempts to disclose the vulnerabilities to one particular vendor resulted in communications with multiple accounts, their helpdesk and the generic feedback addresses, each telling us that security was the job of the other. In the end, we attempted to submit these vulnerabilities to the vendors multiple times over the course of the four month period without any acknowledgment.

Within hours of the paper being covered by the Wall Street Journal [Valentino-Devries 2015], all vendors responded to our previous queries. One vendor claimed to have never received any reports from us, only later to acknowledge their receipt upon further investigation. Vendor responses ran the gamut. Some vendors thanked us for the detailed reports, and a subset noted that they would make the recommended changes to their applications to improve security. One vendor, Airtel, argued that because they moved functionality from the MyAirtel app to a new Airtel Money app, our results were invalid, and insisted that we retract them. However, given that our analysis was correct for that application at the time, we stood by our results and offered to publicly discuss the specifics. Airtel responded by sending us a copy of their newest application and asked us to re-analyze it. Our limited resources prevented us from immediately responding to Airtel's request. More critically, we felt that such a "do-over" was unfair to all of the other application vendors. As such, we decided to wait until all vendors had the time to make the recommended changes and re-evaluate the industry. We believe this approach not only serves as scientific validation of our previous results, but also created an opportunity to measure the effectiveness of our disclosures.

Since publishing the initial paper, we have worked judiciously with stakeholders across the entire mobile money ecosystem. These include regulators at the International Telecommunications Union (ITU), nongovernmental organizations including USAID and the World Bank's Consultative Group to Assist the Poor (CGAP), industry representatives, the United States Department of State and a number of foundations. We traveled to countries including the Democratic Republic of Congo, Mozambique, China, and Malaysia to share our results. All parties expressed great interest not only in our work, but also in the continued growth of mobile money systems as trustworthy institutions. Accordingly, many of these groups asked us to continue our evaluation.

Thus, we waited one year from our first analysis to see how the industry changed as a result of our research.

8. REEVALUATION

We now revisit our analysis nearly one year later to determine whether the state of mobile money has improved. We compare the security of last year's applications with apps downloaded in March 2016 using both automated and manual analyses.

We find that the state of mobile money security for Android is essentially no better than it was in 2015 [Reaves et al. 2015]. In particular, mobile money applications remain vulnerable to TLS man-in-the-middle attacks at four times the rate of normal apps [5] and nearly half only contact servers with insecure TLS configurations. Our preliminary manual analysis showed that the applications we inspected have not repaired critical vulnerabilities in their applications, even while they have continued development of non-security aspects of these applications. Ultimately, more work is needed on the part of researchers, regulators, and organizations, but most of all from developers of mobile money applications themselves.

The following subsections detail our new automated analyses, the results of those analyses, and a summary and discussion of our re-evaluation of the applications we manually analyzed in 2015. In total, this work is the result of hundreds of man-hours of additional analysis.

8.1. Automated analysis

In our re-evaluation, we again use Mallodroid as an indicator of security in mobile money applications. We also use lightweight static analysis to allow us to evaluate server-side configurations of mobile money applications in an automated fashion.

8.1.1. Mallodroid Results. We used Mallodroid to analyze new versions of all apps studied in 2015 as well as an additional 6 apps newly available in 2016. 2 apps from 2015 failed to analyze, 3 apps from 2016 failed to analyze, and 2 apps were no longer available in 2016. We found that of the 40 remaining apps that had results for both 2015 and 2016, 16 apps (40%) remained safe, 12 (30%) remained vulnerable, 4 apps (10%) developed new vulnerabilities, and only 8 apps (20%) corrected vulnerabilities present in 2015. Of the 6 apps new to 2016, 3 had no TLS vulnerability, 1 was vulnerable, and 2 failed to analyze. Ultimately, only 20% of apps that were vulnerable in 2015 were fixed in 2016, indicating only a meager improvement in security. Moreover, mobile money apps are still far more likely to have a TLS vulnerability than Android apps in the typical Google Play market, as reported by Fahl et al. [2012].

8.1.2. Using Qualys. In 2015, we ran Qualys solely on the apps that we manually analyzed. However, in 2016, we ran Qualys on all mobile money apps collected in 2016. It was not practical to manually analyze every app to identify service endpoints, so to approximate the set of servers that apps connect to, we use a regular expression to scan the disassembled class files for strings that appear to be an HTTP URL. We then manually analyze the extracted domains to remove common domains unlikely to be related to the mobile money functionality (e.g., facebook.com). This technique is not able to determine if a domain is actually contacted during the regular execution of an app. For example, we extract a number of testing domains that would only be run in debugging versions of the app and are not accessible externally. This technique could also miss domains that might be constructed programmatically during execution. Nevertheless, this analysis provides insight into the domains contacted by mobile money apps. We next discuss our findings from this analysis.

[5] Compared to the base rate reported in Fahl et al. [2012].

Grade	Worst Qualys Grade (# Apps)	Endpoint Grades (# Apps, not # domains)
A	4 (17%)	14 (61%)
B	3 (13%)	6 (26%)
C	5 (22%)	8 (35%)
F	5 (22%)	8 (35%)
Invalid Certificate	4 (17%)	6 (26%)
No HTTPS Support	2 (9%)	2 (9%)

Table III: 2016 Automated Qualys Results

8.1.3. Qualys Results. After manually filtering common domains unrelated to mobile money functionality, we recovered a total of 91 HTTP domain endpoints[a] from 25 apps in the 2016 app collection. Of those, 11 endpoints indicated they were for testing only, 4 endpoints had no valid domain records, and another 12 were unresponsive to requests. 2 apps had no reachable endpoints, so the remainder of this section discusses an analysis of 23 apps in total.

The automated Qualys analysis provides insight into the state of endpoints. However, because we are examining many endpoints for some apps, each having a potentially different grade, summarizing these results is not simple. For example, averages are not appropriate because an app may have one secure endpoint and another that leaks credentials. However, it could be the case for that same app that most communication is over secure channels. Thus, we analyze the test results in two ways: a "worst case" analysis that considers the lowest grade an app receives, and an analysis that determines the percentage of apps that contact a domain with a specific grade. This second analysis recognizes that most apps are contacting servers with good TLS configurations but may still be vulnerable.

Table III presents these results. Column two shows that only 12 apps (52%) have endpoints whose worst grade is passing ("C" or better), so only roughly half of mobile money apps are certain to use secure TLS endpoints. The table shows in column 3 that while many apps contact safe endpoints, they also contact dangerous ones as well. While 14 apps (61%) contacted an endpoint with a score of "A", 8 apps (35%) also contacted failing endpoints. In fact, 9 apps (39%) contacted both a passing endpoint and a failing endpoint (i.e., graded "F", having an invalid certificate, or that failed to support HTTPS at all). Of the 14 apps with both Qualys and Mallodroid results, we find that 3 out of the 4 apps with failing Mallodroid scores also have failing Qualys TLS configurations.

In addition to our automated analysis, we also reviewed the results from the apps that we manually analyzed. We find that in some cases applications improved in security, but in others no changes were made, while others still had grades that decreased owing to vulnerabilities in TLS that were not known at the time of our first analysis. Both Airtel Money and GCash had grades that decreased from passing to failing because of newly-discovered TLS vulnerabilities. GCash was vulnerable to the LogJam attack [Adrian et al. 2015], while Airtel Money was vulnerable to CVE-2016-2107. MCoin does not even have a Qualys score as it *still* uses an unsafe, self-signed certificate issued to `localhost`. mPay was the only application we tested whose TLS configuration significantly improved (from "F" to "A").

Overall, the low scores in both Qualys analyses are problematic in their own right. In the case of invalid certificates or no HTTPS, it means that no secure communication is likely occurring. In the case of low SSL test grades, it means that services make themselves vulnerable to cryptographic attacks that threaten the safety of the protocol.

[a]Qualys will not scan IP addresses or non-standard ports, so these were removed from the list of endpoints to be analyzed.

ID	Score	Vulnerability	MyAirtel	GCash	mPay	MCoin	Money on Mobile	Oxigen Wallet
		2015/2016 Similarity	*10.0%*	*67.5%*	*46.2%*	*100%*	*9.68%*	*49.2%*
V0	Med.	Sensitive information stored in shared preferences		x	x	x		
V1	Med.	Sensitive information logged in plaintext		x	✓		x	
V2	High	PIN sent via SMS during registration		x				
V3	High	Session ID weakly constructed		x				
V4	Crit.	Symmetric encryption key packaged with application	NEW	x				
V5	Crit.	Disabled TLS certificate validation	✓	x	✓	x		
V6	Crit.	SSL certificate on API server is not trustworthy				x		
V7	Med.	ACRA debug code mishandles user information				x		
V8	Crit.	Uses plaintext HTTP endpoints	✓				x	
V9	Crit.	Fails to authenticate users to the service					x	
V10	High	Low entropy key generated						✓
V11	Crit.	Encryption key generated on device and transmitted in plaintext						x
V12	Crit.	Public key distribution vulnerable to man-in-the-middle attack						x
V13	Crit.	Messages from server to client have no cryptographic protection						x
V14	Med.	Discloses mobile numbers of account holders in network message						x
V15	High	Weak SMS password reset						x
V16	Crit.	Encrypts a user pin by using a weak key	✓					
V17	Crit.	Accepts/performs unauthorized commands from server			x			

✓: Resolved Vulnerability
x: Unresolved Vulnerability

Table IV: Status of Previously Identified Vulnerabilities

However, TLS configuration is just one aspect of operational security, and we speculate that if services are failing to correctly maintain their TLS infrastructure, there will be many other operational aspects that are neglected.

8.2. Manual Analysis

We also repeated manual analysis on updated versions of the applications that were manually analyzed in 2015. We obtained new copies of all the apps from the Google Play market on March 2, 2016. We then decompiled and reverse engineered these applications. Our primary research question was: "did these applications fix their critical security vulnerabilities?" As a result, our analysis focused solely on confirming whether or not these vulnerabilities were still present in these apps. Even with knowledge of the prior app versions, this was still a significant undertaking. We note that all but mCoin had been updated according to the version number listed in the Google Play market. Many of them saw *significant* design changes, meaning that re-confirming the vulnerability was non-trivial.

To better understand how much development occurred on each application, we used the Androsim application similarity tool included in Androguard [Desnos and Gueguen 2011]. This technique uses signatures of methods to determine how many methods two applications share, and produces a measurement of similarity between the apps. We used Androsim to compare the 2015 and 2016 versions of each application. While mCoin was unchanged in our evaluation period, all other applications examined saw *significant* development activity in the form of new code. This new code includes new analytics libraries, changes to user interfaces, and new functionality. The similarities ranged from 67% in the case of GCash to a remarkable 9.7% for MoneyOnMobile. This means that, for example, the 2016 version of MoneyOnMobile shared only 9.7% of methods with the 2015 version. Table IV includes similarity metrics for manually analyzed apps.

A summary of our repeated manual analysis findings are represented in Table IV. This table presents each manually analyzed app and the index of the vulnerability label as defined in Section 4. As shown in the table, we find that the vast majority of applications have taken few, if any, steps to improve their security, despite significant development in other areas. This is the case even after responsible disclosure followed by press coverage, which we detailed in Section 7. This table also indicates a score indicating the authors' subjective assessment of the severity of the vulnerability.

8.2.1. Airtel Money. In 2015, we analyzed the MyAirtel app (package name `com.myairtelapp`) which at that time allowed for administering customer accounts for mobile service as well as provided an interface to Airtel's mobile money product. After we disclosed vulnerabilities in this product, Airtel informed us that another app — "Airtel Money" (`com.airtel.money`) — was now the primary interface to their mobile money platform. In the interest of completeness, we sought to verify old vulnerabilities present in MyAirtel as well as examining Airtel Money for any new vulnerabilities.

MyAirtel has seen a significant update since it was initially analyzed. Androsim indicated that the 2016 app has only 10% similar methods compared to the 2015 version. Among the changes made is a vast reduction in the mobile money functionality of this application: it appears that mobile money function is limited to using Airtel Money for paying one's mobile bill.

In 2015, MyAirtel had 3 vulnerabilities: failing to properly validate TLS certificates (V5), encrypting a user PIN by using a static key (V16), and using plaintext HTTP endpoints (V8). The code written by Airtel concerning TLS is now handled appropriately. However, an included third-party analytics library, MoEngage, contains code that modifies the default TLS socket factory to trust any certificate. Messages to this library include the user's email address and possibly the user's name. This example highlights the dangers of using third party code in security-sensitive applications. Some API endpoints (but not all) used by the main application function have transitioned to HTTPS. For example, the initial request for an SMS verification PIN is sent over an HTTP endpoint (using a custom hybrid encryption implementation), but the actual PIN is submitted over HTTPS. Two other analytics libraries included in the code also used plain HTTP endpoints.

We did not find evidence that MyAirtel is continuing to encrypt the PIN with a fixed key for authentication in the same way that it was in 2015. However, we did find that an analytics library is encrypting the mobile phone number with a static key in communications with the analytics endpoint. Given that no other data is encrypted in this manner, this may be an attempt to obscure an identifier for data storage purposes and not for preserving confidentiality or integrity. However, regardless of intent, this scheme guarantees that a adversary could recover the phone number if desired.

We discovered other issues related to encryption. First is the use of weak cryptographic algorithms. For example, registration information is encrypted using 3DES in ECB mode — not an IND-CPA secure mode like counter or cipher block chaining mode. Other communications contain an encrypted timestamp. It appears to be used to guarantee freshness or as a message authenticator; however, it is not encrypted using an authenticated encryption mode, allowing for possible modification.

Airtel's second mobile money app, Airtel Money, correctly uses HTTPS and TLS. Airtel Money does include code that disables certificate validation, but it is dead code (i.e., never executed) left over for debugging purposes or included in unused library code. However, Airtel Money was not perfect in every regard. Airtel Money uses the same encryption technique that MyAirtel uses when sending the phone number to an analytics server (and the key is the same).

We also discovered another example of misused cryptography in how Airtel Money stores data on the device. Android provides a common mechanism for storing frequently used key-value pairs called SharedPreferences. Because these are saved to local storage and could potentially be read by other applications, Airtel Money encrypts and decrypts the keys and values when writing and reading to Shared Preferences for certain (potentially sensitive) items including the first name, last name, and mobile number of the user. This is a good idea, but there are several problems with the implementation. First, the encryption key is stored *in plaintext* alongside the encrypted data in the Shared Preferences. Thus, any adversary that can read the encrypted preferences can also find the key. In the worst case, the attacker can test each shared preference until finding the one that decrypts the other encrypted preferences. Second, the developers tried to make it difficult to extract this key by obfuscating the preference name used to reference the stored key. The developers compute the preference name by providing the package name (static) and the Android ID (a device identifier available to every app) as inputs to a SHA1-HMAC-based PBKDF2 algorithm. Even if it were not trivial to extract all shared preferences, this obfuscation would provide little benefit as the inputs to the name-generation function are predictable. Finally, the developers encrypt their data using ECB mode, which does not provide IND-CPA security.

8.2.2. GCash. We identified six severe vulnerabilities (V0-V5) in GCash in 2015, including improper PIN storage, logging, and transmission; improper session ID generation and logging; improper symmetric encryption key storage; and disabled TLS certificate validation. In 2016, we re-analyzed the updated GCash application. Although only 67.5% of the 2016 code is similar to the 2015 code, we found that all six vulnerabilities still remain in the app.

Several of the remaining vulnerabilities expose a user to extensive risk. Improper PIN handling can allow an attacker to obtain the user's PIN and perform transactions that can hurt the user, such as transferring money to another account. Since the PIN is the only input required to authenticate the user before completing transactions, this could be detrimental to the user. A session ID is a token passed to the server for authentication. Improper session ID handling can allow an attacker to obtain the session ID and impersonate the user. Ultimately, with the user's identity, the attacker could then complete transactions. An invalid certificate is also harmful to the user. A man-in-the-middle attack can occur, allowing an attacker to manipulate traffic and ultimately alter transactions.

8.2.3. mCoin. In 2015, we analyzed the mCoin app and found four major security vulnerabilities (V0, V5-V7). Those vulnerabilities included disabled TLS certificate validation, a non-trustworthy SSL certificate on the API server, personal data stored in shared preferences, and mishandling of user information by the debug code.

We found that the TLS certificate associated with the app URL was still expired, self-signed, and issued to "localhost" instead of the correct URL. This vulnerability allows attackers to perform a man-in-the-middle attack in order to manipulate traffic and alter transaction details. Also, personal information stored in the app's shared preferences is also harmful. An attacker with local access can read any information written to the local device and in turn steal the user's identity.

8.2.4. mPay. We discovered four vulnerabilities (V0, V1, V5, V17) in mPay during our 2015 evaluation, including sensitive information stored in shared preferences and logged in plaintext, disabled TLS certificate validation, and unauthorized commands to and from the server. We found that two of the four vulnerabilities still exist. The developers addressed V1 by disabling logging. In addition, the developers addressed V5 by pinning the certificate. Improper logging and storing sensitive data can cause leakage of sensitive data.

8.2.5. MoneyOnMobile. In 2015, we found the following three security vulnerabilities (V1, V8, V9) in the MoneyOnMobile app: failure to encrypt messages by using plaintext HTTP endpoints, failure to authenticate users to the service, and leakage of sensitive information in logs. As noted in Table IV, although less than 10% of the 2016 code was similar to the 2015 code, we found that all three of the vulnerabilities still remain.

Unencrypted messages can be read and modified by an attacker. This vulnerability can allow an attacker to change transaction information and steal money from the user's account. Lack of authentication opens the door for attackers to access sensitive information and make sensitive requests, such as sending money. As stated previously, logged sensitive information can be retrieved by an attacker for malicious purposes.

8.2.6. Oxigen Wallet. We found six vulnerabilities (V10-V15) in Oxigen Wallet, including low entropy key generation, improper key distribution, lack of proper message cryptography, improper disclosure of mobile number, and improper SMS password management. In 2016, we found that only one vulnerability was removed from the app (low entropy key generation). However, we discovered that the app now uses a short static string as a seed to generate the new encryption key, making the key predictable. We found that five of the six vulnerabilities were not properly addressed and removed.

Lack of message confidentiality and integrity gives attackers the ability to access and modify messages. Such manipulation could result in an unauthorized financial transaction. Insecure public key management and lack of cryptographic protection on server to client messages can also result in theft. The attacker can successfully launch a man-in-the-middle attack and send an an alternate key to the app. Once accomplished, the attacker will have access to messages sent from the client, which could contain the user's username and password or transaction information. Oxigen Wallet uses SMS for password retrieval. If a message is intercepted, the attacker could steal the app, log in as the user, and steal funds from the user's account.

8.3. Summary

This section presented extensive evidence from automated and manual analyses of apps from across the mobile money ecosystem that despite interest from the security community, regulators, and non-governmental agencies, security improvements have not occurred. Despite extensive development in other areas of their technology, mobile money operators continue to place their users at risk.

9. LESSONS LEARNED AND RECOMMENDATIONS

We now offer recommendations based on our experience over the past two years. We plan to continue to work with all stakeholders to ensure that these lessons are shared and recommendations are implemented.

9.1. Security Must Be Prioritized

We spoke with a variety of stakeholders throughout the mobile money ecosystem over the course of the last year, and many publicly expressed the need for strong security mechanisms. However, a number of vendors indicated that security represented a significant cost to their businesses, especially given the small margins possible per transaction. Some vendors also felt reduced incentive to focus on providing strong security as they believed that their competitors were also not prioritizing security.

It is our position that these beliefs are incorrect for two reasons. First, while traditional banks often employ expensive fraud detection algorithms to identify theft, the security problems and remedies we highlight in this work can largely be addressed without significant additional cost. For instance, correctly handling certificates within mobile applications and configuring TLS properly on servers adds little in terms of overhead. Moreover, certificates can be issued at low or even no charge through a range of Certificate Authority programs such as Let's Encrypt [Internet Security Research

Group (ISRG) 2016]. It is our position that transaction security must be dealt with by all vendors before further security mechanisms like multifactor authentication or fraud detection algorithms can meaningfully be discussed. Second, we note that vendors such as Zuum provide a useful counterexample to the argument that competing systems are not prioritizing security. Because we were able to identify vendors using best practices, we argue that such practices are not only within the realm of technical possibility, but also that competing solutions do indeed incorporate these protections.

Software vulnerabilities emerge as a result of a complex mix of technological, educational, and policy issues. Developers may not be aware of the dangers, they may have misplaced priorities (e.g., preferring to ship a vulnerable product instead of missing a deployment deadline), or temporary code may never be replaced. It is possible that development resources are limited to fix complex problems like poor authentication. As researchers studying the apps, our perspective is limited. However, we see that some applications remain vulnerable despite public vulnerability disclosure and continued development on other aspects of these apps.

We believe that security will only improve in these systems if there is sufficient incentive to do so. Such an incentive can take several forms. For example, public disclosure of security issues in academic literature and in the popular press may put pressure on companies to improve security. Accordingly, we hope that by again bringing light to the issues in this community, our results can be used to help usher all parties towards more secure operations. Another venue is industry self-regulation in the form of standards adopted and implemented throughout the industry. Such self-regulation is attractive in markets where there is a perceived risk of government regulation. We will continue to respond to the needs of the community by taking leadership in issuing recommendations on security for digital financial services within organizations such as the ITU. Of course, carefully implemented and enforced government regulations that mandate secure systems (and/or that makes operators responsible for losses as a result of technical flaws) will also likely improve incentives to develop secure systems.

9.2. Vendors Must Make Reporting Easy

One of the challenges we faced during the first iteration of our study was alerting the vendors of security vulnerabilities. In spite of our repeated attempts through multiple channels, it was not until our work was covered in the press that we were able to engage in a dialogue. Because virtually all software systems have vulnerabilities, a strategy for dealing with future problems should be developed.

Cyber Emergency Response Teams (CERTs) do not exist in many of the regions that benefit most from mobile money systems, and often do not have relationships with every vendor where they are established. Instead, we suggest that vendors take a more proactive step and create a `security@` email address. This standard naming convention will make reporting security issues easier, and will ensure that vendors do not miss reported vulnerabilities.

We are in the process of making this suggestion become reality. As part of the outreach efforts undertaken by the authors, we are working with the GSMA, which serves as the industry group for many of these vendors. Specifically, we are looking to incorporate this recommendation into their industry wide Code of Conduct [The GSM Association (GSMA) 2016].

9.3. Directions for Improving Security

As shown in Figure 1, vendors claim a wide range of security protections in spite of our findings. Without specific training to perform the complex reverse engineering and network analysis conducted in this work, however, few users can validate the truth of these claims. This represents a significant threat to both users and brands, as loss of trust in brands means both decreased revenue and usage. Unfortunately, no platform and few, if any, non-browser apps provide transport security indicators.

While not perfect, we urge mobile platform vendors (e.g., Google, Apple) to develop such indicators. Many mobile browsers now use the familiar lock icons long seen in the desktop space [Amrutkar et al. 2015], but no such standardized option exists for apps. We appreciate that the issue of getting indicators "right" is difficult, and that many users do not understand the full context of a lock icon; however, such information can be used much more quickly by expert users to reason about transport security. It is our belief that having this operating system provided icon would not only move vendors away from their own vulnerable cryptographic protocols to more standard approaches, just as has happened in the traditional Web. Without such an indicator, it is our fear that applications will continue to be able to make security claims with little real evidence.

There are other opportunities to improve the security of these and other applications. One such mechanism is the App Security Improvement Program currently being conducted in the Google Play market [Google 2016]; this system scans submitted applications for common and easily detected vulnerabilities, and requires developers to remediate them. Another possible mechanism is to detect potential vulnerabilities at development time by integrating security linters into common Android development tools like Android Studio. Reaves et al. [2016] examine the limits of Android research and provide additional directions to improve app security.

10. 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 [Anderson 1993]. 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 and Cutrell [2010] present a new scheme that avoids vulnerabilities with using one-time passwords with PINs and SMS. Sharma et al. [2009] propose using scratch-off one-time authenticators for access with supplemental recorded voice confirmations. 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 [Reaves et al. 2016] or message tampering [Nyamtiga et al. 2013b], and so have seen several projects to bring additional cryptographic mechanisms to mobile money systems [Chong 2009; Nyamtiga et al. 2013a; Cobourne et al. 2013]. 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 [Paik 2010].

Several authors have written papers investigating the holistic security of mobile money systems designed exclusively for "dumbphones." Paik [2010] notes concerns about reliance on GSM traffic channel cryptographic guarantees, including the ability to intercept, replay, and spoof the source of SMS messages. Panjwani [2011] 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. While those papers focus on technical analysis, Martins de Almeida [2013] and Harris et al. [2013] note the policy implications of the insecurity of mobile money.

After our initial study, Castle et al. published a complementary study that provided a threat model for mobile money applications, interviewed mobile money developers, and presented results of from a high level measurement analysis of 197 applications [Castle et al. 2016]. Their app analysis provided insights into download counts, permission use, minimum versions, and use of URLs and third party libraries in mobile money applications. They also interviewed seven developers from Nigeria, Kenya, Uganda, Zimbabwe, and Colombia, finding that most developers were aware of security concerns, but found it challenging to find good sources of information about security or struggled with budgetary or partner requirements that limited security efforts. These challenges may inform our technical findings that many vulnerabilities were not corrected after public disclosure.

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 [Enck et al. 2014], a dynamic analysis system that detected private information leakages. Shortly after, Felt et al. [2011] found that one-third of apps studied held privileges they did not need, while Chin et al. [2011] found that 60% of apps manually examined were vulnerable to attacks involving Android Intents. More recently, Fahl et al. [2012] and Egele et al. [2013] use automated static analysis to study 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 using cryptographic libraries. This project is most similar to the work of Enck et al. [2011], 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.

11. CONCLUSIONS

Branchless banking applications have and continue to hold the promise of improving the standard of living of 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. Many of these vulnerabilities remain even one year after responsible disclosure and press coverage. 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 and Jared Trinkler for their assistance with this work. We would also like to thank colleagues at the Florida Institute for Cybersecurity Research (FICS) at the University of Florida for their help in preparing this work, as well as our anonymous reviewers for their helpful comments. We gratefully thank PNF Software for an academic discount on their JEB decompiler.

This work was supported in part by the US National Science Foundation under grant numbers CNS-1526718, CNS-1540217, and CNS-1617474. 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

David Adrian, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Bguelin, Paul Zimmermann, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, and Emmanuel Thom. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. ACM Press, 5–17. DOI:http://dx.doi.org/10.1145/2810103.2813707

Airtel TOS 2015. Airtel Money: Terms and Conditions of Usage. http://www.airtel.in/personal/money/terms-of-use. (2015).

Chaitrali Amrutkar, Patrick Traynor, and Paul van Oorschot. 2015. An Empirical Evaluation of Security Indicators in Mobile Web Browsers. *IEEE Transactions on Mobile Computing (TMC)* 14, 5 (2015).

Ross Anderson. 1993. Why Cryptosystems Fail. In *Proc. of the 1st ACM Conf. on Comp. and Comm. Security*. ACM Press, 215–227. DOI:http://dx.doi.org/10.1145/168588.168615

Apktool 2016. android-apktool: A Tool for Reverse Engineering Android APK Files. https://ibotpeaches.github.io/Apktool/. (2016).

Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Abdulrahman Alkhelaifi, and Kevin Butler. 2014. Securing SSL Certificate Verification through Dynamic Linking. In *Proc. of the 21st ACM Conf. on Comp. and Comm. Security (CCS'14)*. Scottsdale, AZ, USA.

Nick Berry. 2012. PIN analysis. http://www.datagenetics.com/blog/september32012/. (Sept. 2012).

Bill & Melinda Gates Foundation. 2016. Financial Services for the Poor: Strategy Overview. http://www.gatesfoundation.org/What-We-Do/Global-Development/Financial-Services-for-the-Poor. (2016).

Bouncy Castle 2014. The Legion of the Bouncy Castle. https://www.bouncycastle.org/. (Nov 2014).

Bradley Reaves, Jasmine Bowers, Sigmond A. Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. 2016. *droid: Assessment and Evaluation of Android Application Analysis Tools. *Comput. Surveys* 49, 3 (Oct. 2016).

Sam Castle, Fahad Pervaiz, Galen Weld, Franziska Roesner, and Richard Anderson. 2016. 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)*. ACM, New York, NY, USA, Article 4, 10 pages. DOI:http://dx.doi.org/10.1145/3001913.3001919

Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proc. 36th Intl. Conf. Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 175–186. DOI:http://dx.doi.org/10.1145/2568225.2568286

Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proc. 9th Intl. Conf. Mobile Systems, Applications, and Services (MobiSys '11)*. ACM, New York, NY, USA, 239–252. DOI:http://dx.doi.org/10.1145/1999995.2000018

Ming Ki Chong. 2009. *Usable Authentication for Mobile Banking*. Ph.D. Dissertation. Univ. of Cape Town. http://pubs.cs.uct.ac.za/archive/00000520/

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

Sheila Cobourne, Keith Mayes, and Konstantinos Markantonakis. 2013. Using the Smart Card Web Server in Secure Branchless Banking. In *Network and System Security*. Number 7873 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 250–263. http://link.springer.com/chapter/10.1007/978-3-642-38631-2_19

Italo Dacosta, Mustaque Ahamad, and Patrick Traynor. 2012. Trust No One Else: Detecting MITM Attacks Against SSL/TLS Without Third-parties. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, 199–216.

Anthony Desnos and Geoffroy Gueguen. 2011. Android: From Reversing to Decompilation. *Proc. of Black Hat Abu Dhabi* (2011). http://media.blackhat.com/bh-ad-11/Desnos/bh-ad-11-DesnosGueguen-Andriod-Reversing_to_Decompilation_WP.pdf

Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proc. 20th ACM Conf. Comp. and Comm. Security (CCS '13)*. ACM, New York, NY, USA, 73–84. DOI: http://dx.doi.org/10.1145/2508859.2516693

William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2 (June 2014), 5:1–5:29. DOI: http://dx.doi.org/10.1145/2619091

William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A Study of Android Application Security. In *Proc. 20th USENIX Security Sym.* San Francisco, CA, USA. http://www.usenix.org/events/sec11/tech/full_papers/Enck.pdf

William Enck, Patrick Traynor, Patrick McDaniel, and Thomas La Porta. 2005. Exploiting Open Functionality in SMS-capable Cellular Networks. In *Proc. of the 12th ACM conference on Comp. and communications security*. ACM, 393–404.

Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgartner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proc. 2012 ACM Conf. Comp. and Comm. Security (CCS '12)*. ACM, New York, NY, USA, 50–61. DOI: http://dx.doi.org/10.1145/2382196.2382205

Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *Proc. 20th ACM Conf. Comp. and Comm. Security (CCS '13)*. ACM, New York, NY, USA, 49–60. DOI: http://dx.doi.org/10.1145/2508859.2516655

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proc. 18th ACM Conf. Comp. and Comm. Security (CCS '11)*. ACM, New York, NY, USA, 627–638. DOI: http://dx.doi.org/10.1145/2046707.2046779

GCash TOS 2015. GCash Terms and Conditions. http://www.globe.com.ph/gcash-terms-and-conditions. (2015).

Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. 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)*. ACM, New York, NY, USA, 38–49. DOI: http://dx.doi.org/10.1145/2382196.2382204

Google. 2014a. Dashboards — Android Developers. https://developer.android.com/about/dashboards/index.html. (Nov 2014).

Google. 2014b. Log — Android Developers. https://developer.android.com/reference/android/util/Log.html. (Nov 2014).

Google. 2014c. Random — Android Developers. https://developer.android.com/reference/java/util/Random.html. (Nov 2014).

Google. 2016. App Security Improvement Program. https://developer.android.com/google/play/asi.html. (2016).

GSMA 2014. MMU Deployment Tracker. http://www.gsma.com/mobilefordevelopment/programmes/mobile-money-for-the-unbanked/insights/tracker. (Nov 2014).

Andrew Harris, Seymour Goodman, and Patrick Traynor. 2013. Privacy and Security Concerns Assocaited with Mobile Money Applications in Africa. *Washington Journal of Law, Technology & Arts* 8, 3 (2013).

Vaughn Highfield. 2012. 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).

Internet Security Research Group (ISRG). 2016. Let's Encrypt. https://letsencrypt.org/. (2016).

JEB Decompiler 2014. JEB Decompiler. http://www.android-decompiler.com/. (Nov 2014).

Janet Kamana. 2014. M-PESA: How Kenya Took the Lead in Mobile Money. http://www.mobiletransaction.org/m-pesa-kenya-the-lead-in-mobile-money/. (April 2014).

Gilberto Martins de Almeida. 2013. 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).

mCoin TOS 2015. mCoin: Terms and Conditions. http://www.mcoin.co.id/syarat-dan-ketentuan. (2015).

Christopher Mims. 2013. 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).

Money on Mobile TOS 2015. Money on Mobile Sign-Up: Terms and Conditions. http://www.money-on-mobile.com/?page_id=930. (2015).

mPay TOS 2015. mPAY: Terms and Conditions. http://www.ais.co.th/mpay/Condition.html. (2015).

S.J. Murdoch, S. Drimer, R. Anderson, and M. Bond. 2010. Chip and PIN is Broken. In *Security and Privacy (SP), 2010 IEEE Symposium on*. 433–446. DOI:http://dx.doi.org/10.1109/SP.2010.33

Baraka W. Nyamtiga, Anael Sam, and Loserian S. Laizer. 2013a. Enhanced Security Model For Mobile Banking Systems In Tanzania. *Intl. Jour. Tech. Enhancements and Emerging Engineering Research* 1, 4 (2013), 4–20. http://www.ijteee.org/final-print/nov2013/Enhanced-Security-Model-For-Mobile-Banking-Systems-In-Tanzania.pdf

Baraka W. Nyamtiga, Anael Sam, and Loserian S. Laizer. 2013b. 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 (2013), 38–43. http://www.ijteee.org/final-print/oct2013/Security-Perspectives-For-Ussd-Versus-Sms-In-Conducting-Mobile-Transactions-A-Case-Study-Of-Tanzania.pdf

Josh Ong. 2014. 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).

Oracle. 2014. Random - Java Platform SE 7. https://docs.oracle.com/javase/7/docs/api/java/util/Random.html. (Nov 2014).

Oxigen TOS 2015. Oxigen Wallet: Terms and Conditions. https://www.oxigenwallet.com/terms-conditions. (2015).

Michael Paik. 2010. 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)*. ACM, New York, NY, USA, 54–59. DOI:http://dx.doi.org/10.1145/1734583.1734597

Saurabh Panjwani. 2011. Towards End-to-End Security in Branchless Banking. In *Proc. 12th Workshop on Mobile Comp. Syst. and Appl. (HotMobile '11)*. ACM, New York, NY, USA, 28–33. DOI:http://dx.doi.org/10.1145/2184489.2184496

Saurabh Panjwani and Edward Cutrell. 2010. Usably Secure, Low-Cost Authentication for Mobile Banking. In *Proc. 6th Symp. Usable Privacy and Security (SOUPS '10)*. ACM, New York, NY, USA, 4:1–4:12. DOI:http://dx.doi.org/10.1145/1837110.1837116

PCI DSS 2016. Data Security Standard — Requirements and Security Assessment Procedures. https://www.pcisecuritystandards.org/documents/PCI_DSS_v3.pdf. (Apr 2016).

Claire Penicaud and Arunjay Katakam. 2014. *Mobile Financial Services for the Unbanked: State of the Industry 2013*. Technical Report. GSMA.

Qualys. 2014. SSL Server Test. https://www.ssllabs.com/ssltest/. (Nov 2014).

Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R.B. Butler. 2015. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 17–32. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/reaves

Bradley Reaves, Nolen Scaife, Dave Tian, Logan Blue, Patrick Traynor, and Kevin R. B. Butler. 2016. Sending out an SMS: Characterizing the Security of the SMS Ecosystem with Public Gateways. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. San Jose, CA.

Reserve Bank of India 2008. *Mobile Payment in India — Operative Guidelines for Banks*. Technical Report. Reserve Bank of India.

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

Safaricom. 2014. Relax, you have got M-PESA. http://www.safaricom.co.ke/personal/m-pesa/m-pesa-services-tariffs/relax-you-have-got-m-pesa. (Nov 2014).

Ashlesh Sharma, Lakshmi Subramanian, and Dennis Shasha. 2009. Secure Branchless Banking. In *3rd ACM Workshop on Networked Syst. for Developing Regions*. Big Sky, Montana.

Richard Shay, Saranga Komanduri, Adam L. Durity, Phillip (Seyoung) Huh, Michelle L. Mazurek, Sean M. Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2014. Can Long Passwords Be Secure and Usable?. In *Proc. Conf. on Human Factors in Comp. Syst. (CHI '14)*. ACM, New York, NY, USA, 2927–2936. DOI:http://dx.doi.org/10.1145/2556288.2557377

Smali 2014. smali: An assembler/disassembler for Android's dex format. https://code.google.com/p/smali/. (Nov 2014).

The GSM Association (GSMA). 2016. State of the Industry Report: Mobile Money. http://www.gsma.com/mobilefordevelopment/wp-content/uploads/2016/04/SOTIR_2015.pdf. (2016).

The MITRE Corporation. 2014. CWE - Common Weakness Enumeration. http://cwe.mitre.org/. (Nov 2014).

Patrick Traynor, Patrick McDaniel, and Thomas La Porta. 2008. *Security for Telecommunications Networks*. Springer.

Jennifer Valentino-Devries. 2015. Wall Street Journal - Researchers Find Security Flaws in Developing-World Money Apps. http://blogs.wsj.com/digits/2015/08/11/researchers-find-security-flaws-in-developing-world-money-apps/. (August 2015).

Zuum TOS 2015. Zuum: Termos e Condições. http://www.zuum.com.br/institucional/termos. (2015).

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	*Unavailable*	
za.co.fnb.connect.itt	South Africa	500000-1000000	
New apps in 2016:			
ci.orange.sva.om	Côte d'Ivoire	50,000 - 100,000	
com.chickencoders.splash	Sierra Leone	100 - 500	N/A
com.compassplus.mobicash.customer	Mail	5,000 - 10,000	N/A
com.comviva.mmgguyana	Guyana	10,000 - 50,000	✗
com.idea.ideamoney	India	100,000 - 500,000	
ezzeload.ezzetech.com.bkashflexi	Bangladesh	50,000 - 100,000	✗

Table V: We found 54 mobile money Android applications located in 30 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.